12-2017

Configurable Random Instruction Generator for RISC Processors

Krunal Mange
kxm3978@rit.edu

CONFIGURABLE RANDOM INSTRUCTION GENERATOR FOR RISC PROCESSORS

by

Krunal Mange

GRADUATE PAPER

Submitted in partial fulfillment
of the requirements for the degree of
MASTER OF SCIENCE
in Electrical Engineering

Approved by:

Mr. Mark A. Indovina, Lecturer
Graduate Research Advisor, Department of Electrical and Microelectronic Engineering

---

Dr. Sohail A. Dianat, Professor
Department Head, Department of Electrical and Microelectronic Engineering

DEPARTMENT OF ELECTRICAL AND MICROELECTRONIC ENGINEERING
KATE GLEASON COLLEGE OF ENGINEERING
ROCHESTER INSTITUTE OF TECHNOLOGY
ROCHESTER, NEW YORK
DECEMBER 2017

I would like to dedicate this work to my mother, my family, friends and Prof. Mark A. Indovina for the support and guidance during the work.

Declaration

I hereby declare that except where specific reference is made to the work of others, that all content of this Graduate Paper are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University. This Graduate Project is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text.

Krunal Mange

Dec, 2017

Acknowledgements

I would like to thank my advisor, Professor Mark Indovina, for his support, guidance, feedback, and encouragement which helped in the successful completion of my graduate research. Special thanks to my colleagues Namratha Pashupathy Manjula Devi and Thiago Pinheiro Felix da Silva e Lima for their work, corrections, comments and their active participation in this work. Finally, thanks to Dr. Dorin Patru for providing access to student implementations of the various RISC processors discussed in this paper.

Abstract

Processors have evolved and grown more complex to serve enormous computational needs. Even though modern-day processors share same dna with processors half century ago, verifying them today is the huge wall to scale. Verification dominates production cycle even with advances both in software (programming as well as CAD tools) and manufacturing (fabrication) as there are too many test scenarios to cover. Testing complex devices like processors with manual-testing alone in certainty missing the dead lines. Automatic verification is a great way to overcome hurdles of manual testing viz. speed, manpower, and ultimately cost. The work described in this paper targets verification of processors which have in-order instruction execution. Verification is done using SystemVerilog testbench which compares output of device under test to the output of SystemC model, when random instructions are applied.

Contents

List of Figures

List of Tables

Forward

The paper describes a configurable Random Instruction Generator developed as part of a larger Graduate Research project called **Project Heliosphere**. The overarching goal of Project Heliosphere is to develop a robust, configurable, verification and validation environment to further the study of various RISC processor architectures. The initial phase of this project was undertaken by Krunal Mange (Configurable Random Instruction Generator for RISC Processors), Namratha Pashupathy Manjula Devi (Configurable Verification of RISC Processors), and Thiago Pinheiro Felix da Silva e Lima (Reconfigurable Model for RISC Processors). Indeed I am proud, and humbled by the research work produced by this group of students.

Mark A. Indovina

Rochester, NY USA

17 December 2017

Chapter 1

Introduction

High quality verification by directed test-benches is a very difficult task to complete within limited time constraint. Directed test-benches with simulation will get the results quickly but may not cover areas outside the target, if a certain area is missed so are the bugs associated with it. Also directed test-benches do not accurately simulate the real world applications a processor might encounter. Random vector testing helps solve the problem faced by the directed test-benches as instructions are created at random which mimic real world application and if the verification is carried on long enough then it can be sufficiently said as verified. Another advantage of random vector testing is that bugs are found faster, because of the random nature even the bugs not thought of before can be found. The device under test (DUT) in this paper are either 12 or 14-bit in order execution processors which can be of Harvard or Von Neumann architecture. The goal is to build a flexible test-bench such that any of the configurations can be tested and can be scalable to even larger designs. The process is divided into 3 parts viz. random instruction generation, SystemVerilog test-bench and SystemC model. Random instruction generator will generate random instructions depending on the configuration specified in configuration file. Instructions, even though being random, should make sense in relation to spatially neighboring

instructions. Perl is chosen as the language to produce instruction because of its text processing abilities and other features like associative array, etc. Configuration file is also used by SystemC model to calculate expected output for the particular instruction. SystemVerilog is chosen for the test-bench because of its modularity and scalability. A change in test-bench will require only a change in a particular module of SystemVerilog test-bench rather than a complete overhaul.

1.1 Research Goals

The goal of this work is to have a random instruction generator capable of generating instruction for variety of processor which are part of test set and have provisions for other designs. This objective is achieved as:

- Specify the basic requirements for the random instruction generator.
- Classify the processor design variations and come up with steps to create an instruction generator to span all the requirements.
- Incorporate user switches to generate specific category of instruction, which are stored in files compatible with processors and test-bench.
- Generate random instructions and verify two sample processor from set of eight processor, as proof of concept.

A script is also developed to collect data from all the test runs, tabulate the data and generate recommendations for debugging.

1.2 Contributions

The major contributions to the project Heliosphere are:

1. A fully functioning random instruction generator is built in Perl.

2. Error detection and correction for two processors (12 & 14-bits) is carried out by various tests to eliminate all the errors. These two processor samples are considered standard for the rest of the set.

3. Reused a part of instruction generator script to develop result database generator, which organizes and classify errors.

1.3 Organization

- Chapter 2: Research relating to the project outline and technology is presented in this chapter.

- Chapter 3: The test environment overview with brief introduction to each component is discussed in this chapter.

- Chapter 4: This chapter describes in detail processor architecture, instruction set and operation of the processors used in this work.

- Chapter 5: Configuration file structure along with description of each data field is explained.

- Chapter 6: Random Instruction Generator is described in detail in this chapter. This chapter provides a brief overview of the design followed with explanation of design choices and function descriptions.

- Chapter 7: This chapter describes the procedure to get results and gives explanation on how to interpret them. Explanation is supported by relevant data and graphs.

- Chapter 8: This chapter concludes the paper and discusses possible future work.

Chapter 2

Bibliographical Research

An important part of any research project is review current material related to the project specifications and the relevant search results used are discussed in this chapter. The tools needed for the development of intended work are one of the first material under research. The quest is to validate the feasibility of the tools meeting the requirement. After reviewing related work done with similar tools an affirmation is obtained and focus is shifted to the techniques/methods used to accomplish project in processor verification, model design and instruction generation. Languages such as SystemVerilog, SystemC and Perl are selected and used with verificatoin tools by Cadence to accomplish the work.

Processor verification is an intensive task, especially given that increasingly complex designs can be designed and realized via state of the art manufacturing. In order to promote these designs, they must be verified for correctness. Writing test cases for such complex systems manually is quite challenging. If written to exhaustively check the system, this results in quite a large time overhead in realizing the necessary test cases [3, 4]. This is not suitable to market the product or use it in other project as the timeline is missed. This time penalty warrants automated test case generation, which form the basis of the work done in this paper. Various methods are used

to generate instructions/test-cases for processor verification viz. random instruction generation and randomizing a fixed set of test groups. The randomizing of the fixed set of test groups can yield better results if the design has medium complexity since writing all the individual cases will be increasing difficult. Directed manual tests have a short turn around time as they can be easily designed for a particular case. As discussed earlier this method is not suitable for exhaustive testing. Random Instruction testing is in stark contrast to manual testing in terms of setup time. A random instruction generator requires a greater amount of time as randomness in generation has to be constrained to bound tests within system design parameters and not generate meaningless cases while at the same time reaching corners. In order to have best of both worlds some methods researched use a novel way of randomizing part of a test case is done by making selection of opcodes from a table but making the operands randomized [5, 6]. This method gives a short setup time for the first test but will become time consuming as opcode table for all cases need to supplied.

Random instructions can reach all the possible states including the ones not thought by the test designer [7]. These tests when supplied with bias can give an acceptable coverage as against pure random tests [3]. Genetic algorithms are suggested and used in [8] to test a PowerPC architecture. This includes an execution trace buffer giving feed back to the bias generator, a better set of random instructions are generate which force corner cases. Paper [9, 10] discusses generating biases with respect to the instruction groups rather than each instruction to reduce complexity and get better coverage, which is similar to the work done in this paper. The user has capabilities of generating a group of instruction using mode selector discussed in 6.1 of this paper. A different way of generating random instruction is discussed in [11]; this method uses a Linear Feedback Shift Register (LFSR) to generate pseudo-random stream of bits which is given as input to the DUT after being verified as a legal instruction as it is possible to generate illegal instruction with this method. A work similar to [11] is described in [12] which uses graph theory

to estimate test length so that desired coverage is achieved without going overboard with test runs. Paper [13] carries the bias generation a step further with much more customization e.g. test length, instruction biases, memory maps, choice of directing tests to a particular processor element. All this customization helps target tests and also helps while debugging as only a particular area can be identified as a problem and thoroughly tested. The research affirmed the choice of using random instructions generator in this work, as it is versatile and can be automated to give superior testing time.

The test environment in this work is divided into three parts: generator, model and test-bench. This structure is chosen as it is proven, and it can be argued that the generator and model can be combine together like in [14]. But combining the model and generator together introduces negative bias in both as they are dependent. Independently developing the model is more flexible and brings in robustness. The model is written in SystemC as it has the modularity and flexibility of software but can be used to model hardware, were as HDL (Hardware Definition Languages) can do a better job but sacrifice software features of classes, objects [15]. The model is considered as reference in verification and zero errors are expected from it. Various techniques to verify SystemC model are discussed in [16–18] viz. assertion based test, explicit state model test. The flow in verification is that the output of model is considered golden vector and compared to DUT's output, which depending on environment are generated and stored in a file or verified cycle by cycle. The final piece which ties all the parts together in test environment is the test-bench. The test-bench in this paper is developed in SystemVerilog. SystemVerilog is chosen for its versatile nature and re-usability and the test-bench developed in SystemVerilog is scalable as all the parts of the test-bench need not change to accommodate testing of new design [19, 20]. The structure of test-bench is divided as assertion, scoreboard, monitor, interface, driver [21]. Each part is a class and performs a job e.g. driver provides input vectors to the DUT, while scoreboard compare DUT values to golden vector [22]. The work discussed in the paper

also tabulates results and produces graphs, an important observation made from research that presentation of result play equally important role as the testing work itself.

Chapter 3

Test Environment

The test environment houses all the components required for verification. Test environment is divided into three parts Random Instruction Generator (RIG), Model and Test-bench. Random Instruction Generator is written in Perl v5.20 and generates set of instructions in accordance to specifications of the device (processor) under test (DUT). Model is built in SystemC and its function is to generate reference outputs. Test bench is written in SystemVerilog and checks DUT output for errors. Figure 3.1 shows the block diagram of the system.

Testing/verification starts with the configuration file. The tester or owner of DUT populates the information fields in the file depending on the specs. Configuration file in itself is not a functional unit but provides information, viz. processor architecture, register size, number of registers, mnemonic-opcode pair, etc. useful for functioning of the three components. The configuration file is explained in Chapter 5. The RIG generates an instruction set which is read by both the model and test bench. The Model produces ideal outputs and the test bench compares DUT outputs to model's and reports errors if any.

The test environment has eight test samples which are mix of 12 or 14 bits, and Harvard or Von Neumann architecture. Two DUT samples are corrected to have no errors which assisted in

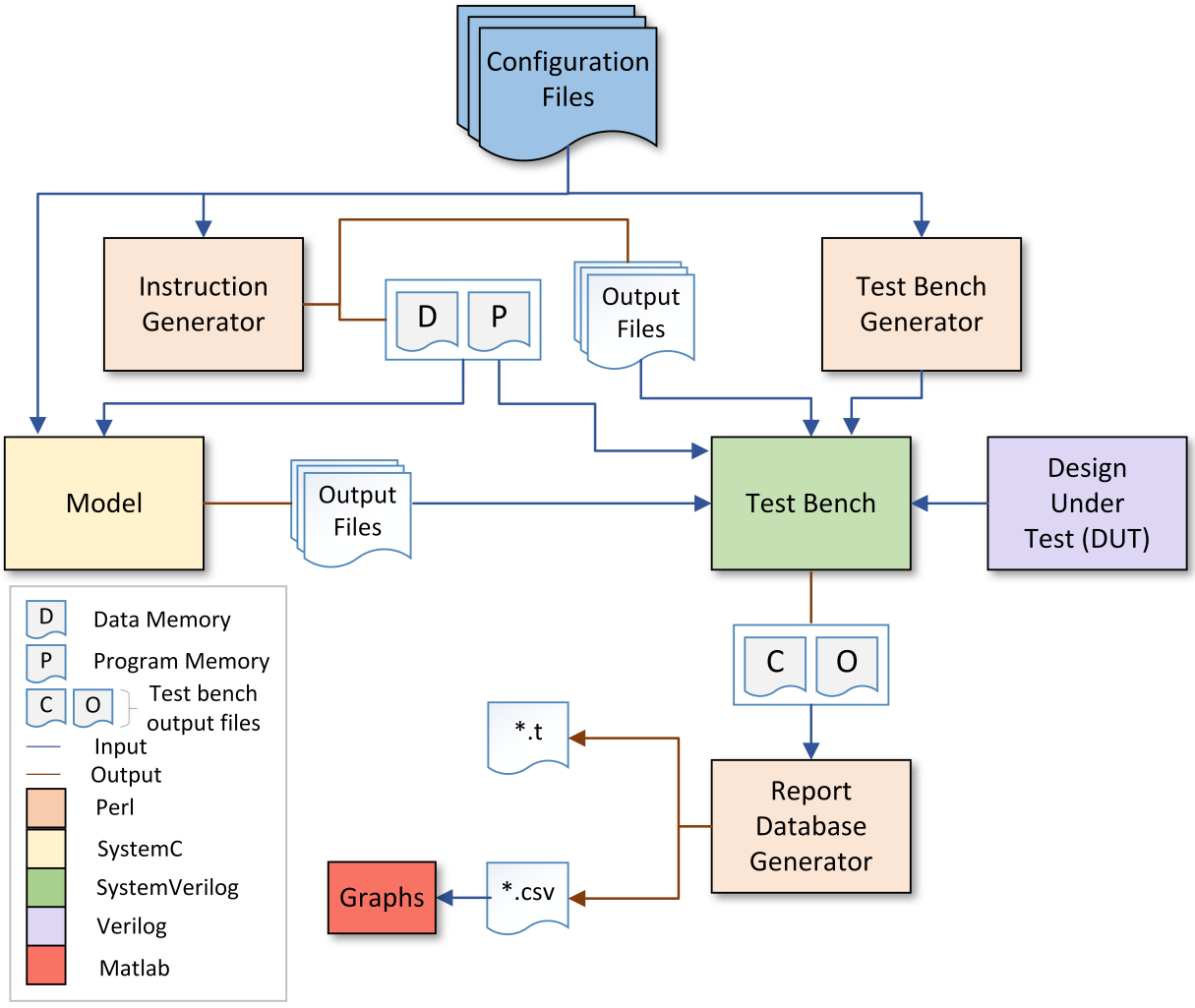

Figure 3.1: System Block Diagram

the building of test environment and is proof of concept. After all the simulations have run, the Report Database Generator (RDG) program cleans and arranges the report in final presentation format.

3.1 Random Instruction Generator

RIG is the first to run. It extracts information from configuration file and reports of any errors in file information. This error reporting helps so that syntax errors are avoided. Depending on information extracted viz. processor architecture, bits, etc, instructions are generated as one or two memory files. For Von Neumann architecture only one memory file is generated with memory size in accordance to configuration file specifications. Whereas for Harvard, two files, instruction & data memory file is created as shown in Figure 3.1. The number of instructions are controlled by the user, if no input is associated with instruction number then a 1000 instruction size is default. RIG also has features which protect against a number of instructions larger than possible to fit, limit number of branch instruction so that the stack doesn't overflow, exception prevention as the DUT's tested could not handle exceptions like divide by zero. Instructions generated can be a mix of all instruction types or any combination like data manipulation, branch - data manipulation, etc. This flexibility eases bug finding as efforts can be focused on a particular instruction type. Since RIG calculates the result of instructions generated to prevent exceptions and control other parameters it also generates files with registers values. The result files aid in environment building as one more checkpoint in testing as extra test vectors.

3.2 Model

The model is built to emulate the processors under the test and is written in SystemC. The model configures itself with the data extracted from the configuration file. Model takes instruction file/s generated by instruction generator and calculates the results. Model writes out the register values in files for each register. Model also generates files with program counter values, which helps test-bench determine if the DUT's flow of execution long is correct, and status flag values. Model checks for exceptions as second safety check after RIG and other errors in instructions if any. Output files generated are input to test-bench as shown in Figure 3.1. The outputs generated by model are compared to the DUT outputs by test-bench [23].

3.3 Test-bench

The test-bench does the verification and reporting of error after simulation. test-bench is written in SystemVerilog due to its modular nature [24, 25]. test-bench is the only common interface between different parts of environment and DUT. The test-bench receives instruction files from the generator and provides stimulus viz. clock, instruction input. This is done by instantiating DUT in test-bench. The test-bench then monitors the output from the processor under test and compares it to the output produced by model and instruction generator as shown in Figure 3.1. Simulation is stopped as soon as an error is detected, a report with expected values for that instruction along with registers values for a set number of instructions is generated. The information about past results help in debugging if the error is in the current instruction or has it stemmed from previous instruction/s. After simulation the result are fed to RDG which complies results and produces tidied report along with 'csv' file with error density of each instruction and potential issues with that particular instruction. This helps in plotting graphs making it easier to interpret the results. test-bench is comprised of environment, test, test case, interface, driver.

DUT resides in the test module. Driver provides stimulus and interface provides data interface between modules. As discussed earlier, the modular nature helps as complete test-bench need not change with change in DUT, this makes it robust and easier to maintain [2, 24]. A Perl script was written to generate the test-bench depending on specs in the configuration file.

3.4 Report Database Generator

RDG is written in Perl v5.20. After required simulations have been ran, RIG creates various logs and reports listing error instruction and other relevant data useful in debugging. The report has opcodes and it is difficult to visually interpret the data. To resolve this RDG inserts in a comment stating mnemonics for the opcodes and makes other changes to make the report more visual. RDG also complies frequency of errors and possible cause from the test-bench report, this data is written to a CSV file making it easier to plot graphs and observe data. RDG is based on RIG as information extraction from configuration file is the same.

RIG is discussed in the paper. Model and test-bench are part of collaborative work to develop the configurable test environment for verification of RISC processors.

Chapter 4

Processor Architecture

The RISC processors used in the work are developed as part of course EEEE621 - Design of Computer Systems supervised by Dr. Patru at Electrical Department, Rochester Institute of Technology in Fall 2015. The specification were unique to each student according to number assigned. All the processors were originally developed in Altera's Quartus Prime using Verilog or VHDL.

4.1 Overview

The processor architecture overview is seen in Figure 4.1. It contains Memory Unit, Registers and Arithmetic & Logical Unit (ALU). Input/Output peripherals are memory mapped to the highest 16 locations. The processors are classified into two types viz. Harvard & Von Neumann architecture. Further sub classification is done on basis of the Instruction Word (IW) size which can be 12 bits or 14 bits wide. Table 4.1 lists all the processors used in the work with their major specifications.

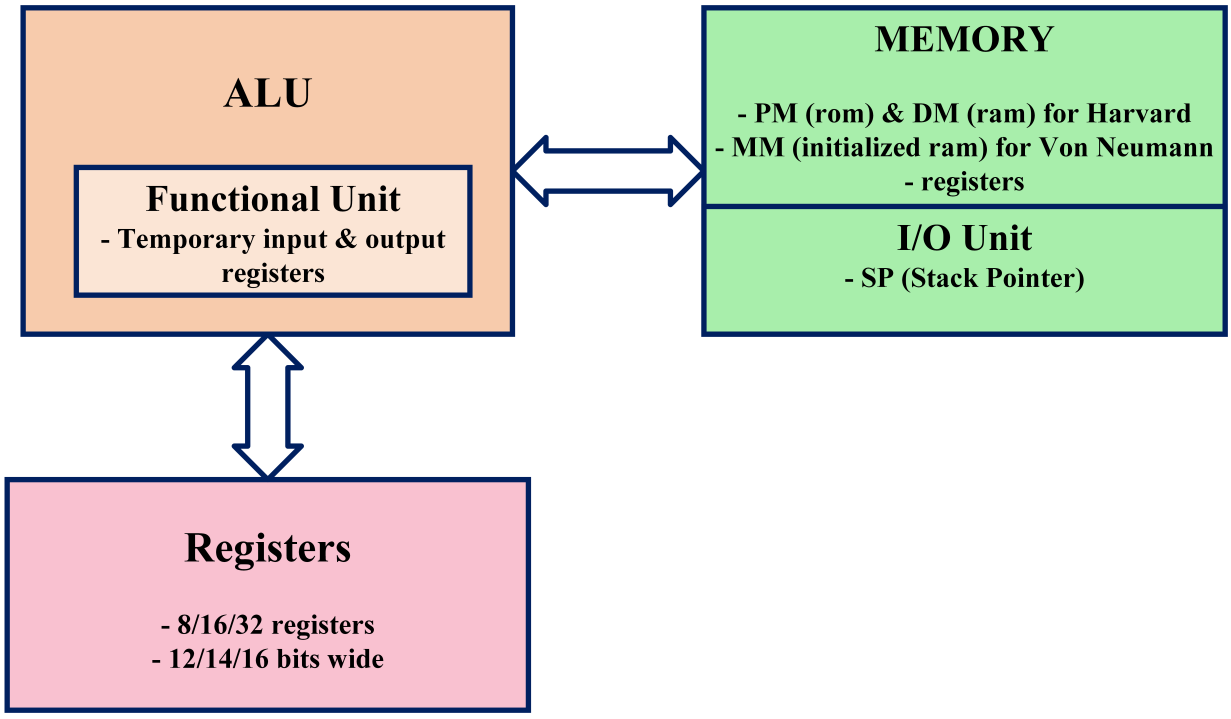

Figure 4.1: Processor Architecture Overview

Table 4.1: Processor list

Processor name	Architecture type	Instruction word size	No. of registers	Status reg. size
paRISC621pipe_v	Harvard	14	16	8
kxmRISC621_v	Von Neumann	12	8	12
vxkRISC621_v	Von Neumann	14	16	8
axtRISC621	Von Neumann	12	8	12
tflRISC621_v	Harvard	12	8	12
dnm_RISC621_v	Harvard	14	16	8
dxpRISC521pipe_v	Harvard	14	16	8
nxpRISC621pipe_v	Von Neumann	14	16	8

4.2 Registers

Registers are used in various operations are performed and they also hold results. Depending on instruction word size, the number of registers is limited as bits required to access them are constrained. All data manipulation instructions have registers as operands and for write back/storage. A 12-bit processor has a 3-bit register field giving eight registers (R0 - R7) whereas a 14-bit processor has a 4-bit register field giving sixteen registers (R0 - R15). A design exception is made for the 12-bit design in flow-control instruction that register1 (operand1) size is limited to 2-bits making 4 registers usable. This is done in order to accommodate four flag bits while still having the same number of opcode bits.

4.3 Arithmetic & Logic Unit (ALU)

ALU is host to mathematical, logical operations, and functional unit. For most of the processors used, mathematical operations and logical operations like shift, rotate, OR, etc are written by the designer. Whereas multiplication and division operation are used as a unit which is generated using Quartus's IP wizard. The ALU in the processors used is not designed to handle exceptions e.g. divide by zero. Functional unit houses temporary registers for both input and output. Input operands are held in temporary registers TA and TB which can indicate register used or can hold constant depending on the instruction. Outputs after an operation are held in temporary registers TALUH and TALUL to hold upper half and lower half of the result respectively. Number and types of operations are same across all processors except that data size handled can be different.

4.4 Memory Unit

The Memory Unit is monolithic for Von Neumann design as both program and data memory reside in same memory space. Harvard design on the other hand uses separate memory space for program and data memory. An advantage of Harvard design is that both program and data memory can be accessed simultaneously. This helps when the instructions are pipelined to increase throughput. Section 4.6.1 discusses pipeline in detail. The memory units used are generated using Quartus's IP wizard. In case of Von Neumann architecture design, initialized ram is used. Initializing ram is required because program and data memory share space and the program needs to be loaded before start of operations. The registers like Program Counter (PC), Stack Pointer (SP), MAeff (Effective memory register) are also a part of memory unit as seen in Figure 4.2. Program counter keeps track of instruction to be fetched and is incremented every cycle except for a stall in pipeline. MAeff is calculated as addition of MAB (Memory Address Base register) and MAX (Memory Address Index register). MAB holds the address offset and MAX holds index which depends on an addressing mode used in a particular instruction [1]. Stack pointer points to top of stack and is incremented or decremented depending on direction of growth. Stack in this work is defined as percentage of memory, it is made even in size by Random instruction generator as Flow control instruction occupy two location on stack. MAeff is register not available to user, it holds the address to the location in the memory from which data can be loaded or stored or jump location. The organization of stack is seen in Figure 4.3.

Input/Output (I/O) peripherals are memory mapped for both the architecture types. The highest 16 locations in memory (data-memory for Harvard) are assigned to I/O peripherals. The peripheral assignment in I/O memory is user defined. A generic memory structure is as shown in Figure 6.3 the table represents Von Neumann architecture but Harvard is also similar with an exception that it has separate program memory.

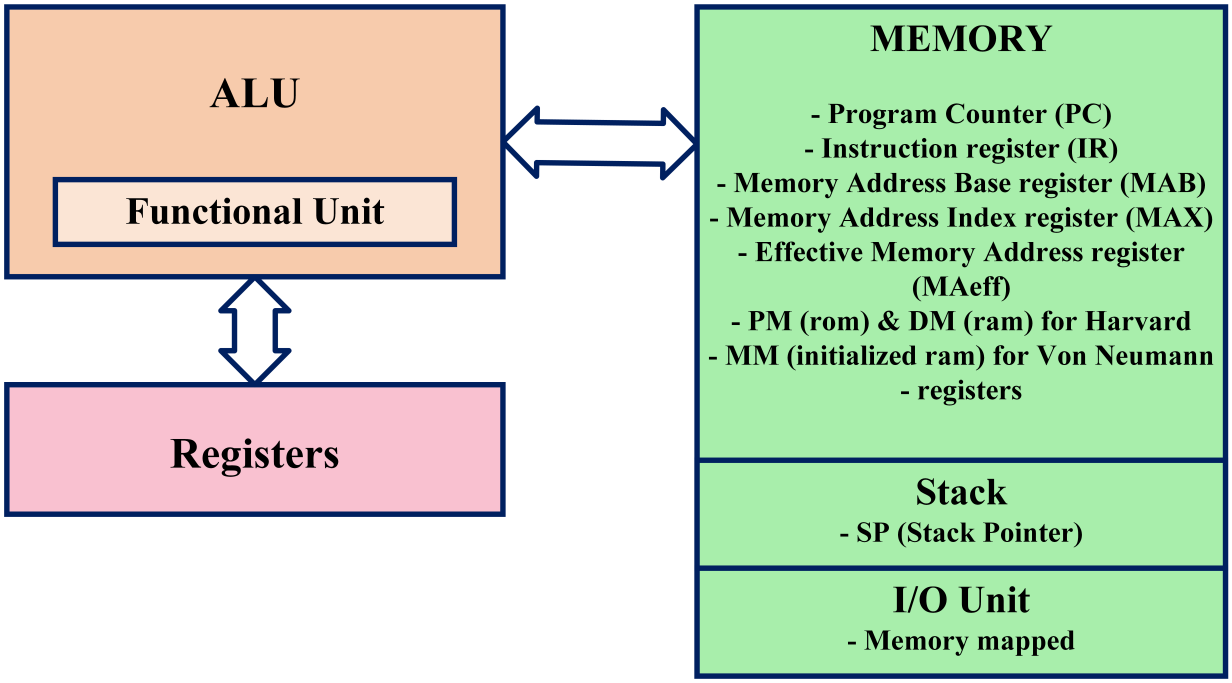

Figure 4.2: Memory organization

4.5 Instruction Set

The size of opcode restricts the number of instructions possible. The processors tested in this work have a 6-bit opcode field giving a maximum of 64 instructions. The actual number of instructions implemented is smaller than 64. Addressing modes define how the data is fetched or stored. For the processor used in the work operand1 (Ri) signifies addressing mode. A value of '0' in the field of addressing mode indicates direct addressing mode that the offset address present in next instruction word is the address to either jump, load or store. A value of '1' indicates, the final address is addition of program counter value and offset address. The rest of the values possible for operand1 field represent register addressing mode. In register addressing mode the final address is calculated as addition of value present in register (pointed by value of operand1) and address offset. The instruction set is classified as follows.

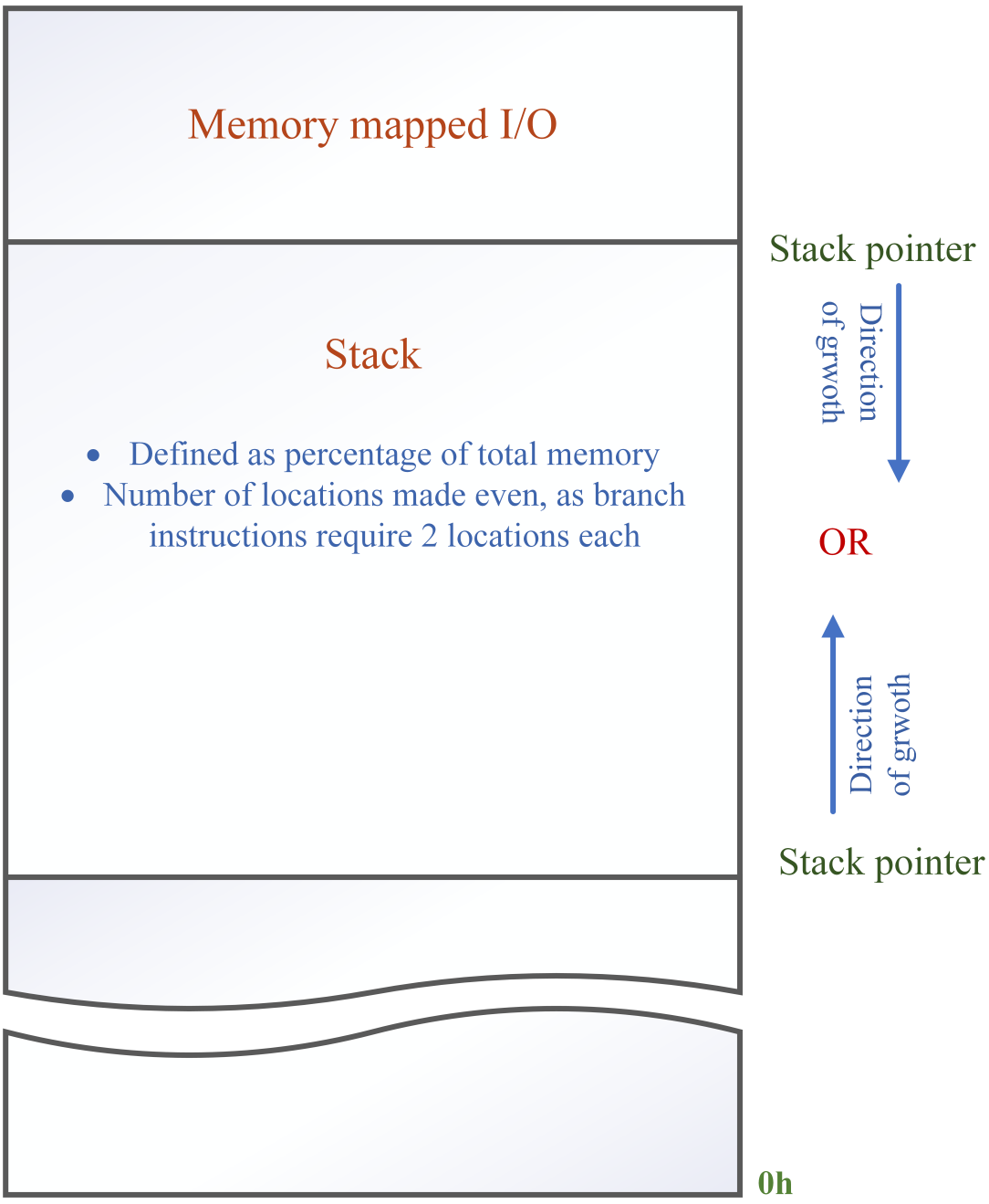

Figure 4.3: Stack Organization

4.5.1 Manipulation Instructions

Manipulation instructions modify data in registers. They are used to perform mathematical and logical operations. They can be further divided in to types i.e. two operand instructions and one operand plus a constant type instruction. Instruction word format for manipulation instructions is as seen in Figure 4.4. As seen in Instruction Word format register Ri is operand1 and stores result as well, Rj is second operand which can be a register or a constant value.

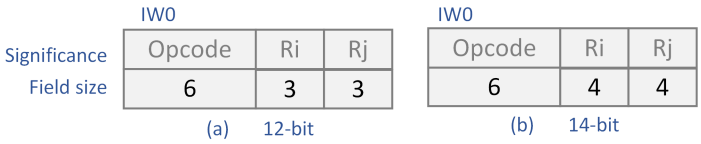

Figure 4.4: Instruction word format for manipulation instructions (a) 12-bit and (b) 14-bit processor [1]

4.5.2 Data Transfer Instructions

Data Transfer Instructions transfer data to and fro between registers and memory. Since operations are register based they are likely to be loaded with value from the memory at start of program and when a value in memory is required. These instructions are also helpful when number of values being held are greater than number of registers, memory here provides temporary storage for the value. Instruction word format for data transfer instructions is as seen in Figure 4.5. The offset address is stored at next memory location and is denoted as IW1 where IW0 would represent the data transfer instruction. Depending on type of addressing mode, IW1 can be the address of location itself or has to be added to Rj register value to obtain final address.

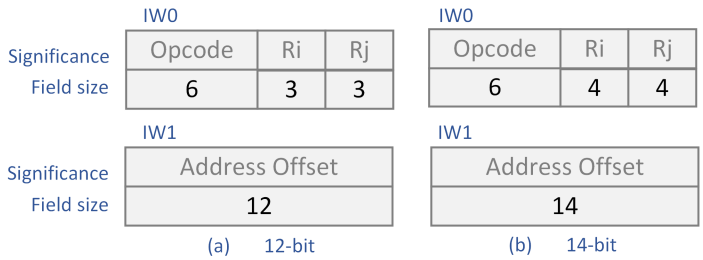

Figure 4.5: Instruction word format for data transfer instructions (a) 12-bit and (b) 14-bit processor [1]

4.5.3 Branch Instructions

Branch instructions help skip or go to a different section of a program. 'JUMP' is a branch instruction which helps skip/jump over set number of instruction forward or backward depending on the condition. If the condition is met the jump is taken whilst if the condition is evaluated to be false, program continues sequentially. The conditions in 'JUMP' instruction are set using conditional flags. Carry (C), Negative (N), Overflow (V), Zero (Z) are four flags used as the conditions. With four one bit flag eight unique conditions plus an additional condition when all the flags are reset. When all four flags are reset then 'JUMP' is unconditional and is taken.

'CALL' and 'RET' (return) are grouped together and are unconditional branch instructions. Call instructions is used to execute a subroutine which is stored after program memory or other part of program memory. Subroutines make programs neat as a piece of code to be executed multiple time need not be replicated at the usage, instead call can be made to the piece of code. Another use possible is to call a exception handling routine. As exception can occur sporadically they cannot be incorporated in main program but stored else where in memory and be executed by call the routine. Branch instructions except 'RET' have IW1 like data transfer instructions. And the address to call routine or jump is calculated using IW1 and Rj register value depending

on addressing mode. 'RET' doesn't have IW1 or address offset as it functions to return flow of program back to caller i.e. to instruction before which call was made. 'RET' instruction restores the values of PC and flag values stored by CALL instruction on stack during its execution. Instruction word format for branch instructions is as seen in Figure 4.6 and 4.7.

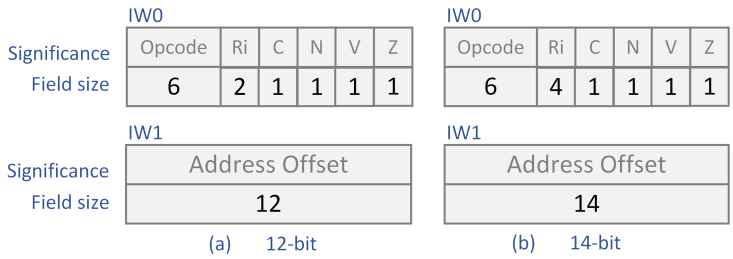

Figure 4.6: Instruction word format for JUMP and CALL (a) 12-bit and (b) 14-bit processor [1]

Figure 4.7: Instruction word format for RET (a) 12-bit and (b) 14-bit processor [1]

All the instructions part of processor architecture are listed in appendix I and are grouped by instruction types.

4.6 Processor Operation

Processor Operations are carried out in machine cycles. Machine cycle is a part of overall operation carried out as per instructions. An instruction operation is considered to be comprised of four parts: instruction fetch, decode, execute and write back in the processors part of the work. These four operations are called machine cycles. Instruction fetch is machine cycle 0 (MC0)

which fetches the first and next new instructions from memory pointed by program counter. Program counter has single increment after every fetch. All the fetches from the memory need not be instructions as depending on type of instruction second instruction word can be memory offset. Instruction decode is machine cycle 1 (MC1) in this cycle depending on instruction opcode (i.e. after identification of instruction) operands are loaded with values from register or constant values. It is in this step that the values loaded in operands can be forwarded from previous instruction to resolve dependencies. Detailed explanation of dependencies is in section 4.6.1 Pipeline Theory. Instruction execution is machine cycle 3 (MC2) where the result for the instruction is calculated. The calculated result is stored in an internal temporary register ready to be written back in next cycle or forwarded if required. Write back is machine cycle 4 (MC3) and is last machine cycle marking end of an instruction cycle. Here the calculated result is either stored in the memory or the register depending on the instruction.

Processor operation is divided into machine cycle to facilitate pipelining of instruction which is similar to a product assembly line. Pipelining increases throughput of processor by utilizing resources efficiently.

4.6.1 Pipeline Theory

Pipeline comes from assembly line used in automobile manufacturing. In manufacturing if a person does all the steps by himself, the steps are performed in sequence and steps ahead have to wait for previous ones to complete. Hence a job with 'm' distinct parts with each part taking 'n' amount of time will take a total of 'mn' time without pipeline. In the same example if we have 'm' people doing an individual task then for first whole job to complete it'll still take 'mn' amount of time. But after first complete job the next job will be completed in 'n' amount of time. This is because after completing first step first person can start working on second job's first part while second person is working on second part of first job. This goes so on and so forth such that

when nth person is working on nth part of first job, the first person is working on first part of nth job. In processors machine cycle replaces a person in completing job i.e. instruction execution. The time taken to fill the pipeline which in example meant first person doing first part of nth job is taken only once if the pipeline is not stalled. Here in processor, a stall can occur when the processor is waiting for an instruction to complete which uses memory and in consequence the memory cannot be accessed.This is a problem especially with Von Neumann architecture as there's a single memory unit and both read-write operation cannot happen at same time. This is an example of structural dependency where multiple instructions try to access same resources. Structural dependencies are solved by stalling the execution of next instruction until resources become available. If there is change in flow of program then the pipeline is reset i.e. flushed. Flushing means completing last instruction before branch instruction then continuing on from new address location where pipeline needs to be filled again. Hence a pipeline is useful when program don't have many branch instructions and such programs appear more often than ones with lots of branches which is why most modern processors use pipelining[26, 27].

A sample example of pipeline is as seen in Figure 4.8.

	Instruction	Result
1	XOR R7, R7;	R7=0;
2	XOR R6, R6;	R6=0;
3	NOT R7;	R7=0xFFFF;
4	ADDC R6, 0x3;	R6=0x3;
5	CPY R7, R6;	R7=0x3;
6	MUL R7, R6;	R7=0x0; R6=0x9;

Figure 4.8: Sample example for pipeline

The example contains six instructions and it takes nine cycles to complete execution. The number of cycles is nine because it takes three cycles to fill the pipeline, six cycles to execute and three trailing cycles to complete remaining instruction execution. Fetch cycle is performed at every clock event and hence not considered as a cycle on its own. Detailed machine cycle for entire execution is seen in Figure 4.9. As seen in sample program, first two instruction reset values in register six and seven. Instruction number three (NOT instruction) which begins at third machine cycle has decode and operand fetch at next cycle but the first XOR instruction has not yet written back the value to R7.

MC	1	2	3	4	5	6	7	8	9
P-IC	XOR-IC	XOR-IC	NOT-IC	ADDC-IC	CPY-IC	MUL-IC	-	-	-
P-MC0	XOR-IF	XOR-IF	NOT-IF	ADDC-IF	CPY-IF	MUL-IF	-	-	-
P-MC1	-	XOR-OPF	XOR-OPF	NOT-OPF	ADDC-OPF	CPY-OPF	MUL-OPF	-	-
P-MC2	-	-	XOR-EXE	XOR-EXE	NOT-EXE	ADDC-EXE	CPY-EXE	MUL-EXE	-
P-MC3	-	-	-	XOR-WB	XOR-WB	NOT-WB	ADDC-WB	CPY-WB	MUL-WB
D/H	-	-	-	RAW	-	RAW	RAW	-	-

Figure 4.9: Pipeline stages

This creates a data dependency and it's of the type Read After Write (RAW) as NOT instruction can read the data only after XOR does write back. This can be resolved by a stall in pipeline, but elegant way of handling this is data forwarding. Data forwarding implies that required data is brought to the required register even before the previous instruction completes write back. With this pipeline continues without a stall. Other dependency example is of Copy instruction. During machine cycle six when 'copy' instruction is in decode cycle it requires data of R6 value for which is being calculated. So data is forwarded from ALU to the register before write back cycle even begins. Other types of data hazards are Write after read (WAR) which occur when instruction over-writes data which is need by one of the previous instruction which needs old data to complete it's execution and Write After Write (WAW). WAW occur when two instructions write data to same register, this happens when instruction are out of order in execution in reference to their issuance.

In the example with six instruction and four cycles each so the program would have taken twenty-four cycles to complete. But with pipeline and no branch it is completed in nine cycles. The speedup of program can be calculated as [total time taken without piepline] / [total execution time with pipeline] [1]. In the example taken the speedup is 24/9 which is 2.66.

Chapter 5

Configuration File

The configuration file is key to the test environment. It contains information to setup test bench, generate test instructions and for model to do calculations. The configuration file is populated from the specification sheet of DUT or by the supplier of the DUT. All the numeric values are specified in the hexadecimal format except for the percentage value. The following information is contained in the file:

name_folder: This field gives directory name in which the files for a particular processor reside. This is relative address to the directory test bench and resides one level below.

name: Name of the processor under test, helps locate the highest level file in directory. This name is also used to generate instruction file, logs and reports under same name which aids accessibility when testing multiple DUT's.

bits: Size of instruction word for the DUT and is also the size of the registers. In the tests done this is also the size of data bus and has two variants 12 bits and 14 bits.

registers: The number denotes the number of registers available to be used in instructions.

architecture: Two types of architecture viz. Von Neumann & Harvard are indicated as, '0' & '1' respectively. This helps generator decide whether to generate a single or separate program & data memory file.

opcode_size: Size of the field dictates number of maximum possible instruction mnemonics. Also helps in generating instruction word and decoding in model.

operand1_size & operand2_size: This denotes the operands max size. For register-register instructions it gives register number used, it also can denote constant value if second operand is constant. For the flow control instruction in processors used, Ri (operand 1) is calculated as: (Instruction-word size) - (4 + opcode_size).

dm_size: This field only applies to Harvard architecture as it has separate data space. The memory length is calculated as (2^dm_size).

memory_size: Total memory size available is indicated by this filed and calculated as (2^memory_size). For Von Neumann architecture this gives available size of program & data memory combined.

pc_in_pc_relative: This information is useful in operations using program-counter(PC) relative addressing mode. DUT design allows relative address to be calculated with respect to current or next instruction address. A '0' indicates current instruction's address is used while '1' indicates next instruction's address is used.

SP & Stack_direction: Top of stack is given by this value. Stack direction indicates the growth of stack i.e. where next element is stored. '0' indicates that the elements are stored from lower to higher address whereas '1' indicates elements stored from higher to lower memory address. The highest the stack can grow or started at is (total memory - 16 highest

locations), this is due to the fact that 16 locations are reserved for Input/output Peripherals (I/O-Ps) which are memory mapped for processors tested.

Stack_size: It is percentage of total memory available, in case of Harvard its w.r.t data memory size. The size is made even as usual push on stack is two words for the processors tested.

Mapping: Mapping of mnemonics is done before specifying opcode value for given mnemonic. Mapping is done between delimiters 'start_mapping' & 'end_mapping'. This allows the test environment to be DUT independent in naming the instruction mnemonic and be flexible in handling various designs. The mnemonics on the left are associated with DUT and the ones on the right are environment specific which are fixed for to be used by instruction generator.

opcode: Here the opcodes relative to the mnemonics are to be entered. Opcodes are divided into three categories data transfer, manipulation & branch instructions each has its delimiter so that the instruction generator can identify different instruction and generate tests with different combinations of instruction types.

clk_st: This value suggests number of clock cycles taken to fill the pipeline or the clock at which first output is obtained. This is particularly useful for test bench to set reference to compare model's output to DUT's for testing purposes.

DUT files:

- name_pm: This file gives the name of program memory used by DUT. This file is present in both Harvard & Von Neumann architecture. For the latter its both program & data memory.
- name_dm: Data memory used by only Harvard type DUT.

- name_div: Divider used by the DUT.
- name_mul: Multiplier used by DUT.
- name_cnt: Counter used by DUT.
- All these files are used by the processor under test to run and perform its operations successfully. These are also required by the test bench to list in compilation file so that the environment instantiate or brings in correct file for to run test.

del_ld & del_st: This field provides information about whether load & store instructions are stalled. '1' indicates stall after the instruction and '0' indicates no stall. This is used by test bench to make result comparison at correct time clock. In usual case only Von Neumann architecture requires stalls as instruction and data are in same memory which introduces one clock cycle latency. Design of Harvard architecture can also stall if designer chooses to do so.

EOF: End of file delimiter.

Example configuration file is shown in Appendix II.

Chapter 6

Random Instruction Generator

RIG generates instructions used for verification and also writes the values of registers to the file. Instructions generated at random help find bugs quicker as the tester need not manually think of combinations. The ability to test with random instructions improves on the test case generation time, if done manually. Hence from early stages of verification bugs or errors in implementation can be found speeding up the turn around time. Random instructions can also test cases which were not thought of if done manually. Manually testing all cases is both labor and time intensive process. With multiple runs of verification, coverage close to 100 percent can be achieved with help of the random instruction generator. The general flow of random instruction generator is shown in Figure 6.2.

6.1 Random Instruction Generator Overview

The RIG is written in Perl v5.20. Perl has powerful text manipulation capabilities while syntax is similar to C. Text processing helps in retrieving data from configuration file and in writing instruction data, register values to file. The text formatting in file is also easier. The instruction

```
perl gen_vt.pl -config filename -mode mode_type
```

Figure 6.1: Command to generate random instructions

Table 6.1: Modes for instruction generation

Mode	Instructions
a	All three types of instruction are generated viz. manipulation, data transfer & branch
m	Only manipulation instruction are generated
mb	Manipulation and branch instruction are generated
md	Manipulation and data transfer instruction are generated

generator is made configurable so that as the input configuration changes, it adapts to it. To achieve this robustness the instruction generator is kept modular and the calculations are kept generic and depend on the input configuration file. The instruction generator also has built in error reporting. If the input configuration file has error in input data syntax or is missing data field then the user is notified of such error with prompts on how to fix the errors. The command to run the instruction generator is seen in 6.1:

The minimum input requirement is a configuration file 5, as it contains all the vital information to proceed with the instruction generation. The other input parameters are number of instruction and 'mode'. RIG can generate user defined number of instructions. The default value for number of instruction is 1000 in this work, to ensure that even without being supplied with number of instruction the RIG works. The input of 'mode' is to compare generating a certain type of instructions, table 6.1 describes 4 modes. The flow of RIG is seen in Figure 6.2. The process begins with extracting information from configuration file done by Extract function. The information extracted gives memory limits and information on segments inside memory like stack, I/O space. With the processor architecture known the next function, 'memory_set', knows about memory being only data or with program memory as well. So the 'memory_set' func-

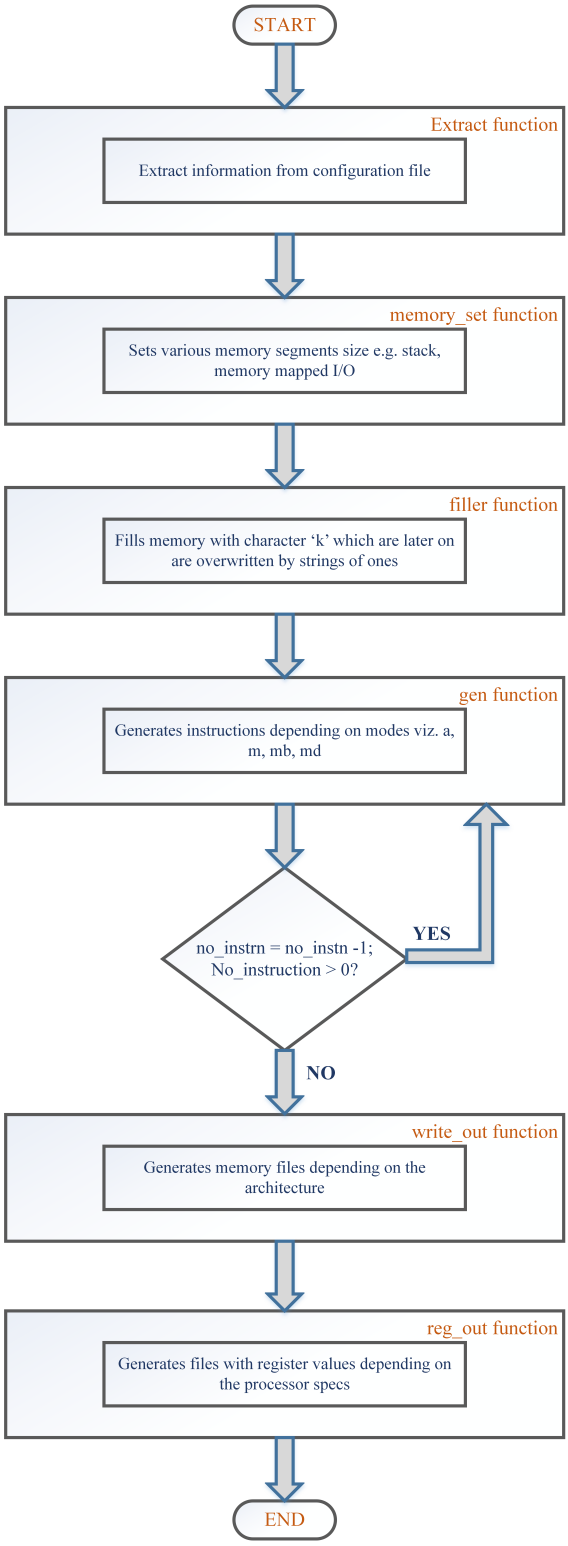

Figure 6.2: Flowchart for instruction generator

tion sets the memory limits for next functions to continue the operation. The 'filler' function fills the memory with zeros in locations which will contain data later on as the instructions are generated. It also fills memory locations with ones which are not written on by generator. The 'gen' function is responsible for generation of instructions. It does so by calling sub-functions viz. manipulation_gen, data_transfer, branch. Instructions generated are written to the file by the 'writeout' function. 'reg_out' function does the task of writing out values of registers to file to be used by testbench. The functions shown in Figure 6.2 are discussed in detail in following sub-sections.

6.2 'extract' Function

The 'extract' function takes the configuration file as an input and extracts all the information about a particular processor. All the values inside the configuration files are hexadecimal, while calculation in RIG are in decimal. Hence all the values are converted into decimal. The values are extracted line by line from the configuration file. Any line beginning with '#' is considered a comment and is ignored. All the configuration file reside in the 'configuration' directory which is one directory level below the RIG. All the extraction of information is done via Regular Expressions (regex) [28]. The regex match and extract a part of a string depending on the criteria given. Mapping of instruction and opcode for mnemonics are written between delimiters e.g. for data transfer instructions 'start_data_transfer:' and 'end_data_transfer:' are the start and end limiters respectively. The delimiters helps in determining a section of data and hence the sections can be in any sequence in configuration file. Absence of either opening or closing delimiter causes an error so that errors in configuration file are avoided. 'EOF' marks the end of file and it is indicator to stop reading data.

6.3 'memory_set' Function

The 'memory_set' function sets up the memory. The function first calculates the maximum memory from the information in configuration file. The value of field 'bits' from the configuration file is taken as power to base 2 to calculate maximum memory possible from that bit size. Here in this work memory for a processor is calculated with its bit size. For Von Neumann architecture only one memory with the calculated size is created, whereas for Harvard architecture two memory with same size are created to serve as program and data memory. But the random instruction generator has provision to support user defined memory which can be smaller or greater than calculated memory size. This custom memory size can be supplied for both program and data memory, if present.

The stack is given as percentage of total memory size or data memory size for Harvard type processors. While calculating memory locations to be reserved for the stack care is taken that the value is even. An even value is chosen as instructions like CALL (branch instruction) use two stack locations one for storing flags and program counter value. To verify the supplied number of instructions can be generated reserved memory is calculated. If the space in memory after subtracting reserved space is greater than desired number of instruction only then RIG proceeds further. In case of instruction number exceeding the memory space available a warning is reported along with the memory space available and execution of generator is carried with maximum memory space possible. Reserved memory is calculated as addition of stack memory, I/O space if memory mapped, subroutine space, data space in case of Von Nuemann architecture and two additional memory locations. The subroutine space in this work is 40 memory locations but can be changed. Data space is reserved in case of Von Neumann type processors so as to allow instruction to write and read from memory location which would be impossible to do if instructions occupied all the space. This reservation is not required in Harvard architecture as

there is separate data memory. Two memory location are reserved as those locations contain one's stored which serves as end of program for testbench at which simulation can be stopped and is completed. At the end of this function memory space is divided accordingly and instructions can be generated. A typical memory layout for Von Neumann type is shown in Figure 6.3.

6.4 'filler' Function

This function serves as building error checking in memory while generating instructions. It fills the arrays with character 'k' which hold data to be written to memory to both program memory and data memory or just to one memory in case of Von Neumann architecture design. The character is chosen at random and it is written as visual aid. Since no address value or opcode at the location will have character 'k' it helps catch error in generated instructions if any. Later on during write out to file this helps in filling memory space with ones where 'k' is found. The locations with ones as data, can be address offset if preceded by a associated valid instruction opcode. At other locations a string of ones mean that the location is not defined. Also this string of ones can help detect branch error when the instruction word reads a string of ones instead of a valid instruction.

6.5 'gen' Function

The 'gen' function is the largest function written. It is responsible for generation of instructions and also prevention of exceptions as the processors in this work exclude exception handling. The 'gen' function is also made modular so that the development and maintenance in future is relatively easier. The instructions are randomized in this function. The mnemonic and opcode of instruction is stored as hash table by 'extract' function. A random opcode from a set of hash

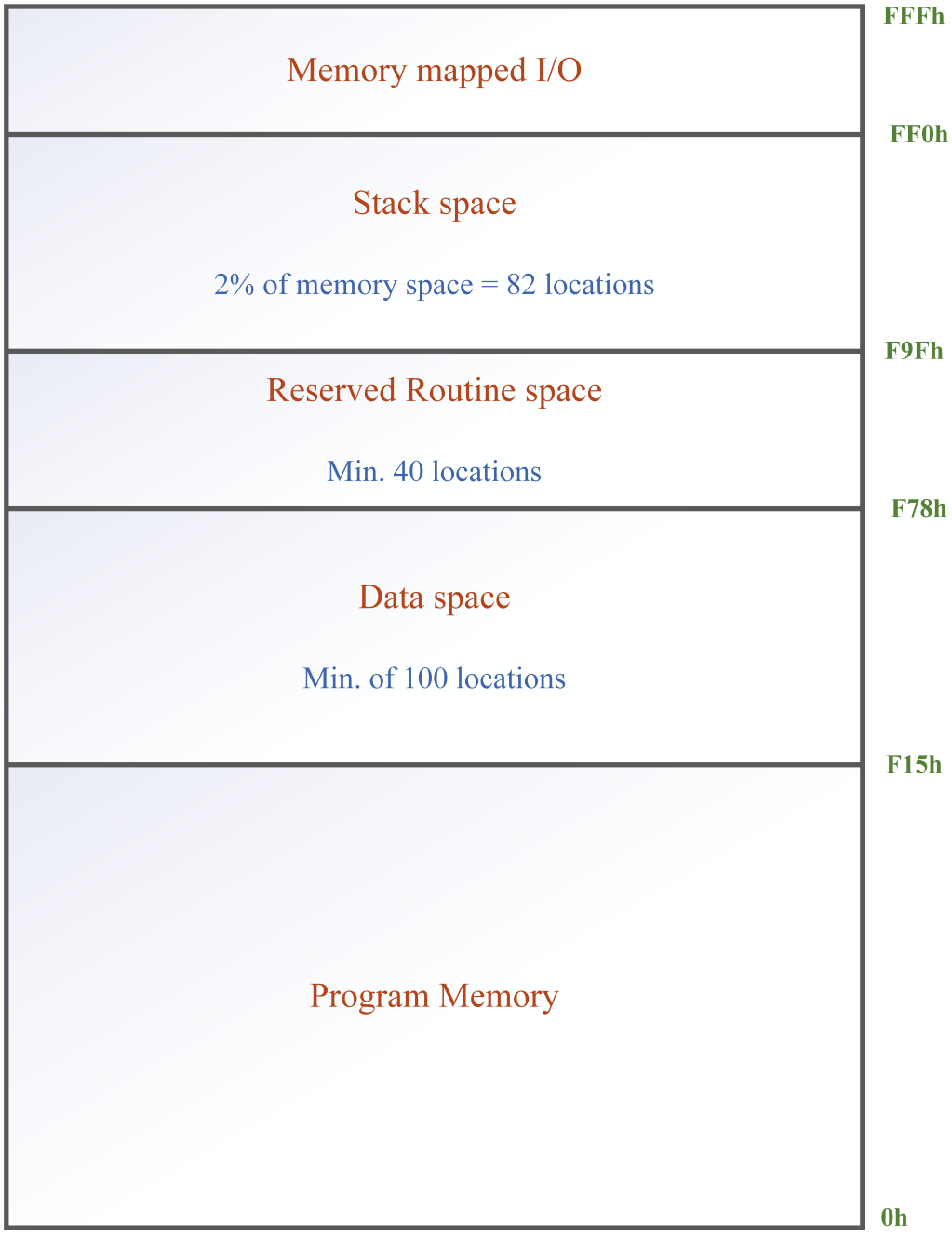

Figure 6.3: Memory Organization

keys is selected via 'shuffle' function which is part of Perl's 'Util' library which contain general purpose subroutines. The 'gen' function is also responsible for generation of a certain set of instructions depending on user input, with 'a', i.e. all instructions, being the default option. A set of only branch or data transfer instructions is not generated as branch instructions need some instructions to branch to. And just branching the flow of execution will not reveal any dependency error which are caused by other instructions when in a combination with branch instructions. Similarly data transfer alone will not yield much information about errors, as reading or writing garbage value cannot be verified if the same arbitrary value is written and read back again. Hence both branch and data transfer instructions are paired with manipulation instructions to provide data to read or write as well as good variation for dependency check when branching. The ability to generate instructions with only certain desired types help in debugging a desired part of the processor design as errors reported will be exclusively from the desired types of instructions. The function is also intelligent to detect if a branch is to an undefined location. In such event branch operation is skipped and replaced with one of the manipulation operations. The data transfer is not chosen in case of an invalid branch because even data transfer from that location will be undefined. After choosing a random instruction 'gen' calls sub-function to generate manipulation, data transfer or branch instructions.

6.5.1 'manipulation_gen' Function

This function to generates manipulation instructions and calculates results for the generated instruction. This function then applies another layer of randomization by randomly selecting registers for the operation. In case of an instruction with constant as the second operand, the register number generated for operand2 still applies as the field size is same. After calculating result of instruction flags need to be set according to the result. The flag assignment is done by 'CNVZ' function, a function is chosen as its functionality is needed by all the instruction and hence it

prevents code duplication. Another function 'compactor' resizes the result to fit the width of instruction word of a particular processor. The resizing of the result is required for certain instructions as the entire result is not stored to destination, some information is contained in flags. An example of such result resizing is adding two 12-bit operands in a 12 bit processor yields a 13-bit result of which the most significant bit (MSB) is carry. Thus result is clipped to 12-bits while 'CNVZ' function already has carry information. The last function to be called to generate opcode for the instruction generated is 'iw1_gen'. The 'iw1_gen' function generates instruction word in format specified by configuration file for a processor. The opcode (instruction word) generated is stored in an array later to be written in memory. Other information stored is resultant register values.

6.5.2 'data_transfer' Function

The 'data_transfer' function also generates random values for the operands. In the case of data transfer instructions, i.e. LOAD & STORE, operand1 represents the addressing mode. Operand2 is the source or destination of data when the operation is STORE or LOAD respectively. The data transfer instructions are two instruction words wide for the processors under the test. The first instruction word which defines the instruction is followed by the address offset. The final address is obtained as an addition of the address offset and base register address (MAB). The base register address is defined by the addressing mode which is the value of operand1. A value of '0' represents direct addressing mode which implies that the final address is equal to the value of address offset. A value of '1' represents PC relative addressing mode and the final memory address is calculated as sum of PC value and address offset. The values from two onwards to the highest number specified by operand1 field (e.g for 12-bit processor it is 7) represents register addressing mode. The final address is equal to the value held by the register. To generate the offset address, RIG generates a random number in valid memory range as the final address. Then

according to the addressing mode, MAB is subtracted to give the offset address. It is ensured that STORE doesn't happen to a program memory location in Von Neumann architecture. This makes sure that the program is not corrupted. If the final memory address is out of memory bounds then the previous value of operand1 (addressing mode) is changed to one that'll fit the memory range. Also since data transfer instructions are allowed in subroutines and before JUMP a caution is exercised result of which the instruction is skipped if only one memory space is available at the end of routine or before jump. A provision is also made to put a random value if a load operation is from a location which is undefined. At the end of the 'data_transfer' function a call is made to 'wi1_gen' to generate the instruction word.

6.5.3 'branch' Function

With a branch instruction selected by 'gen' function the 'branch' function takes over to perform checks and to generate an instruction. Just like data transfer instructions, jump location or subroutine call addresses are calculated according to addressing modes. For JUMP instruction a check if the jump is within the program is performed if it is register addressing mode. If the jump is outside the program, then other registers are checked for providing the address value within program space. One of the other addressing modes is chosen if the previous check fails. For direct and PC relative addressing mode jump size is fixed depending on total number of instructions. Depending on the jump size, the address offset is calculated. The jump size is fixed so that long jumps are avoided which would lead to skipping most instructions making overall test shorter and would increase test time as testing all combinations will take more number of tests. Jump can be taken in a forward and backward direction. A backward jump is of particular interest in instruction generation as it must contain a forward jump to jump ahead of backward jump's instruction location else it'll result in infinite loop. Hence to generate a backward jump a combination of three jumps is generated. The first jump takes flow ahead of second jump which

is a exit jump to avoid infinite loop. The third jump, i.e. the backward jump, takes flow before the second jump which jumps ahead of it.

The 'call' instruction can have its subroutine length as maximum of 10 instructions if it's at top level in nested calls. The nested call can have subroutines with maximum length of 6 instructions. This is done because subroutine space is limited and maximum instruction variation should be checked in a test. A depth level of three nested calls is allowed in this work, depth level can be increased with change in the code. It is made sure that there is a 'return' instruction at the end of a subroutine block to take the flow of execution back to the main program or a subroutine higher in nested call. The 'branch' function also calls 'iw1_gen' to generate the instruction word.

6.6 'write_out' Function

The write_out function writes the instructions generated and the register values for those instructions which were stored in arrays to files. The instructions generated and corresponding resultant register values are then stored in an array as it makes accessing the data and manipulation easier. Also, writing a single value in a file would mean having write handle (pointer) open during the entire writing process increasing risk of data being mishandled. For Von Neumann architecture only one file named 'memory.t' is created and is written into. For Harvard architecture this file also serves as program memory and 'data_memory.t' is the data memory file. These memory files are created in directory named 'heliosphere' which is one directory level below and hosts all the directories for the entire verification environment.

6.7 'reg_out' Function

The register values are written in same directory as RIG by the 'reg_out' function. The file name is same as the register name e.g. for register 0 its 'R0.t' , for stack pointer it is 'SP.t'. All the locations for memory or register may or may not have data in them, those locations are filled with binary string of one. A separate function is created because the decision to write the register values out was made later in project to help with verifying the model's output. This extra set of values helps ensure that model gave correct values while building the test environment. Also with a separate function there is freedom to remove the functionality if required.

Chapter 7

Results

The results are obtained after running simulations. The Report database generator (RDG) then makes visual changes to the reports and generates '.csv' files to facilitate graphing. During first run of simulations, status flag related errors are observed to dominate. Thus majority of simulation runs are cut short and other errors are not reached. This mandated a second test with status flags changed to follow a standard according to the specifications. Test two yielded variety of errors which give helpful insight into design errors and dependencies. The results of two sets of tests, T1 and T2 are collected and graphs plotted to give visual representation and overview of problem area in DUT design.

The RDG is also equipped with feature to classify error into four categories viz, status flag, Program counter, Implementation errors and undefined registers.

Status_Flag: These are the errors in which the status flag don't match the model's output. These errors are prevalent as some design constraint in course EEEE621 - Design of Computer Systems were left at programmer's discretion, which resulted in variations in status flag implementation.

Program_counter: A mismatch in values of program counter are reported under this category.

A wrong branch results in this error, the cause can be wrong memory address calculation or incorrect return address.

Implementation_Error: When a register value doesn't match the expected value it is considered as implementation error. The probability of the instruction's functionality is very high if there is mismatch between expected and observed register value. A few cases where the error may not be implementation related are when a branch is taken to an incorrect location or a return to an incorrect memory location. But this field give a good start in debugging.

Undefined_reg: This type of error is recorded when a don't care ('x') logic value is observed. The usual cause of this error is uninitialized register value on reset or wrong data size transactions between registers.

The above classification of errors are a helpful aid in debugging process as a start point. An Excel spreadsheet file is created to group data, obtained from test-bench, in a table. This excel file is used to create graphs using Matlab, presented in IV. The table has a dependency field related to two operands i.e. Ri, Rj. This field helps identify total occurrences on which an error coincided with a possible register dependency. This being not a perfect number, can give insight into a possible register dependency not being resolved in the design. Another important observation presented to user is total occurrence of an instruction versus total number of errors of that instruction. An example of a test results in tabular form is in table 7.1 which is test T1 in mode 'a'.

Two set of tests are run for a DUT. Each test set has a subset of four test corresponding to four test modes as explained in 6.1. So a total of 8 tests are run for a DUT, each test has 1500 iterations with 1000 instructions each. An example of a script to run 1500 test for a DUT in mode 'a' is as follows:

```
1 #!/bin/bash
```

```
2  rm ../risc/Test_Result.t
3  rm ../risc/cplog.t
4  rm ../pl/perl.log
5  rm log.t
6  cp ../configuration/configuration_2_kxm.txt ../configuration.txt
7  perl ../test_gen/test_genr.pl -len 4 #call to testbench generator
8  for i in {0..1499..1}
9  do
10         cd ../pl/
11         perl gen_vt.pl -config configuration.txt -mode a > perl.log #call to
               instruction generator
12         cd ../processor/
13         make
14         cd ../risc
15         ./sim.csh -r -ng -sv -run
16         echo "Test $i"
17
18 done
19         cd ../pl
20         perl error_rpt.pl -config configuration.txt -mode a -num 1500
21         #call to Report database generator
```

Listing 7.1: bash version

The results are grouped into four categories depending on mode. Each mode has two tests T1 & T2, T1 being results with original DUT design and T2 with status flag modification. Results for a processor 'kxmRISC621_v' are presented and discussed in following sections.

7.1 Mode 'a'

In mode 'a' all the instructions are generated and it gives an overview of errors in the design. This mode can be used at the start to find the group of instructions with most errors and target simulation to that group using different simulation mode. This mode is also useful when most of the errors are cleared and a final sweep can be made to catch errors if any.

7.1.1 Mode 'a' - Test T1

As discussed earlier this test is with no corrections to the status flags. From figure 7.1 it is seen that the status flag error are present for majority of instructions. Errors related to implementation errors are prevalent as for 'Mul' & 'Mulc' as well as 'Div' & 'Divc' instructions the design saves the result in a different way than the standard defined in this work. Other implementation errors are due to an instruction incorrectly implemented e.g. SHLA implementing left shift incorrectly. It can be seen that with help of the figure representing errors it is easier to start debugging . Figure 7.2 shows total occurrence of a particular instruction versus errors for that instruction, this gives a overview of a test set as debugging an instruction with low total occurrence having similar number of errors as an instruction which has more occurrence can be prioritized. The data represented in Figure 7.1 & 7.2 is shown in table 7.1.

7.1.2 Mode 'a' - Test T2

This is the test after making the status flag correction in the DUT. A comparison can be made with test T1 while looking at Figures 7.3 & 7.4 that the number of errors except for status flag error increase. This observation is important because it indicates the differences in the DUT design and the standard considered in this work.

7.2 Mode 'm'

In this mode only manipulation instructions are generated. Generating manipulation instructions exclusively helps catch errors faster as they form a major part of the instruction set. Also since most of these instructions are used in combinations, finding dependencies and eliminating them decremented the time required to find errors in other type of instructions.

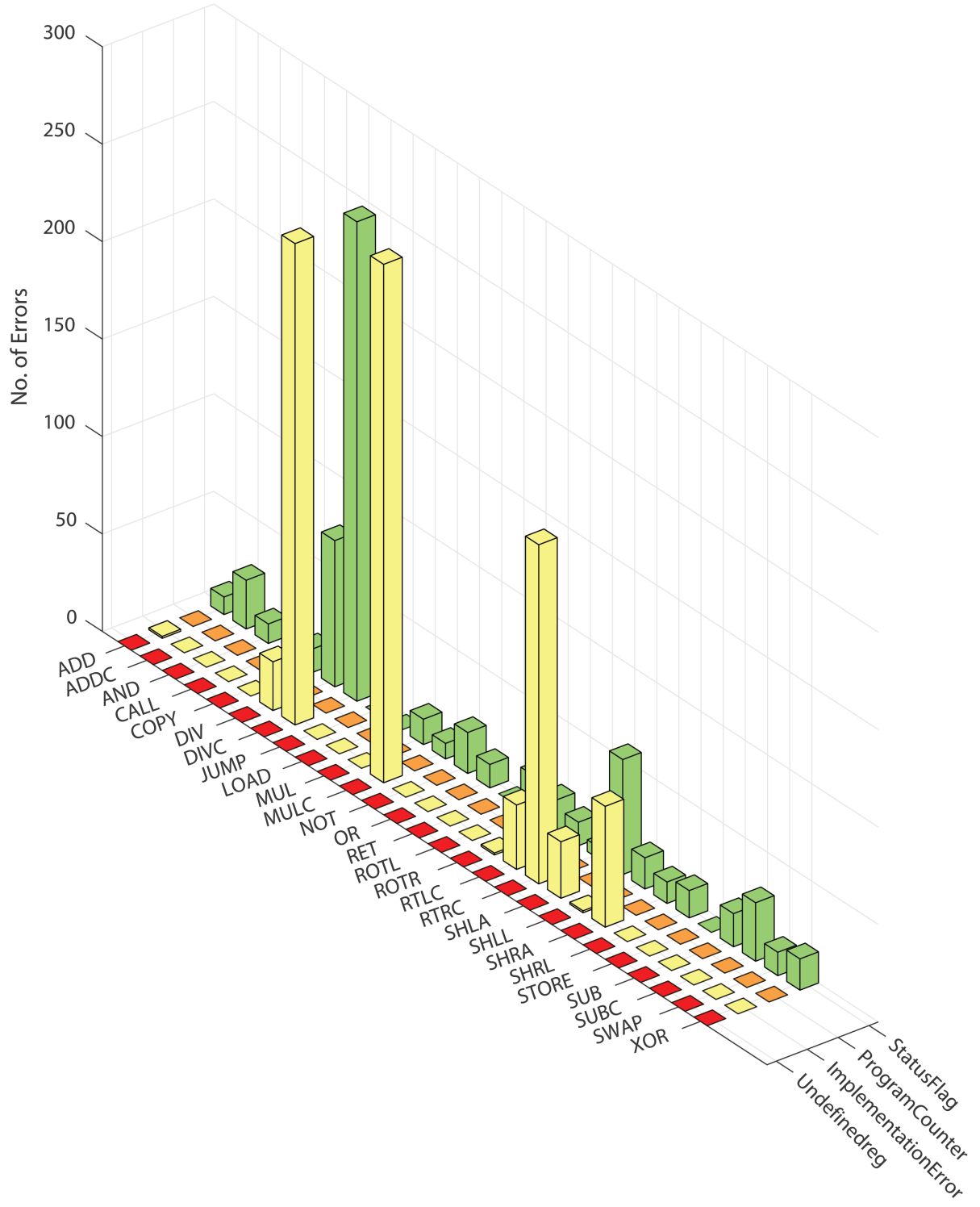

Figure 7.1: Errors for Test T1 - mode 'a'

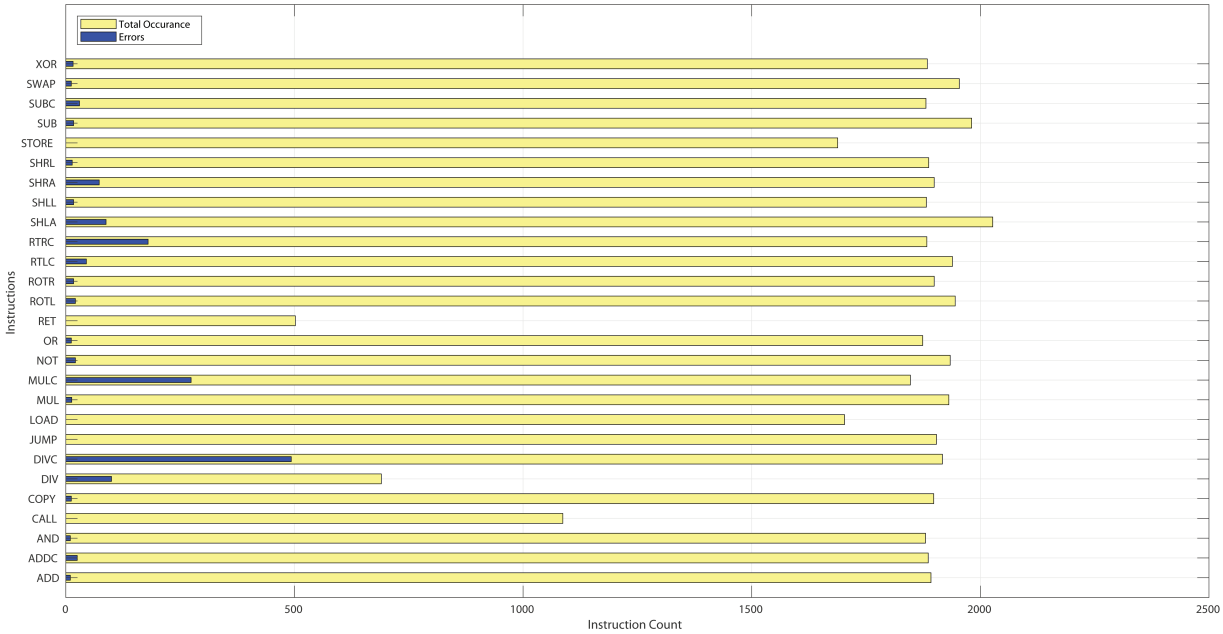

Figure 7.2: Total error count for Test T1 - mode 'a'

7.2.1 Mode 'm' - Test T1

In this test the status flag errors are present, as well as possible implementation errors in instructions like 'DIVC', 'RTRC', 'SHRA'. For test T2 a guess can be made that the status flag error should almost be zero while only having implementation errors as T2 is carried out after correcting status flag error in design. The classification of errors and density of errors is seen in Figure 7.5 & Figure 7.6 respectively.

7.2.2 Mode 'm' - Test T2

From Figures 7.7 & 7.8 it can be observed that the status flag errors are zeros except for 'SHRA' because the implementation is incorrect.

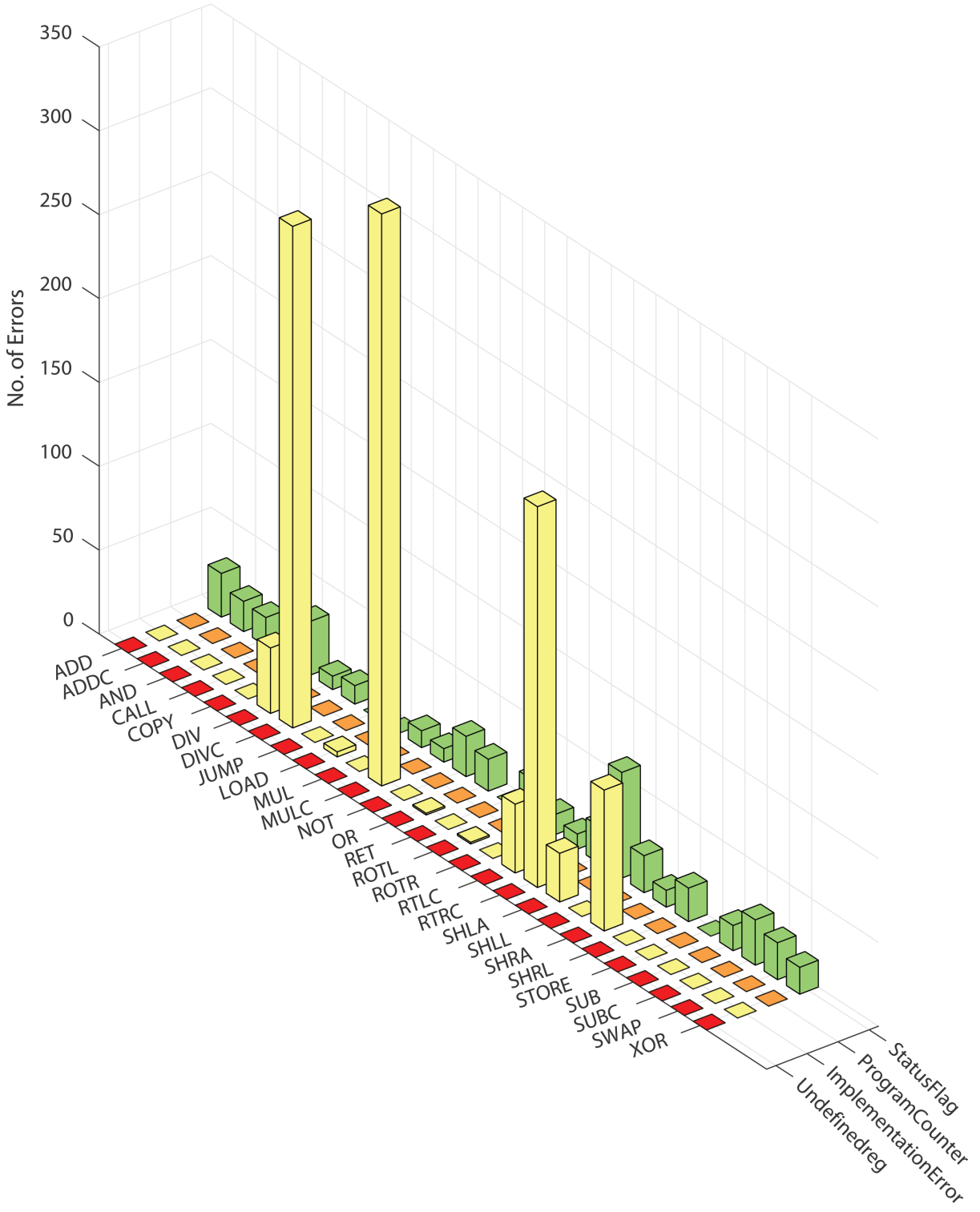

Figure 7.3: Errors for Test T2 - mode 'a'

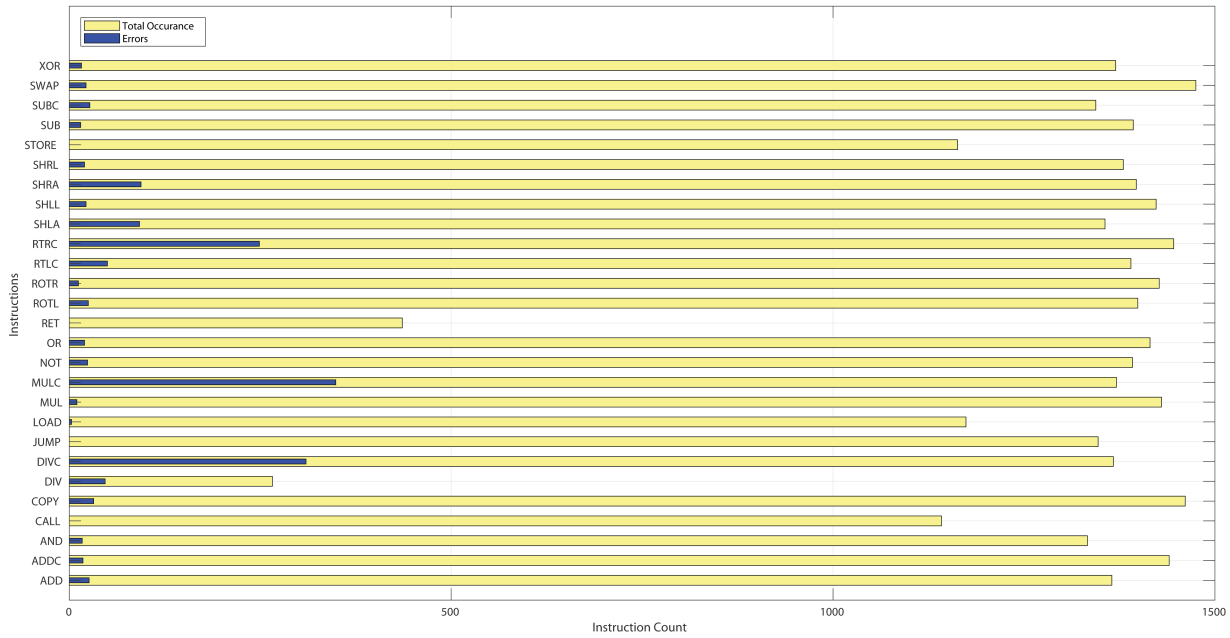

Figure 7.4: Total error count for Test T2 - mode ’a’

7.3 Mode ’mb’

Manipulation and branch instructions are generated and tested in this mode. Errors in branch instructions may or may not be reflect in the register values for that instruction, for example a ’return’ instruction might pop a wrong value from stack into status flag resulting in status flag error. Usually these errors can be detected by comparing expected program counter (PC) values to the observed PC value. Hence an error in PC value is a good indicator for branch instruction related errors. The Table 7.1 prepared by the RDG has column which list PC errors. Errors related to PC can be found for manipulation instructions, but this is usually when the previous instruction being ’CALL’, ’JUMP’ or ’RET’ resulting in wrong branch location. Another good indicator to spot a incorrect branch is don’t care values, this is categorized under ’Undefined reg’ in the table, found in simulation.

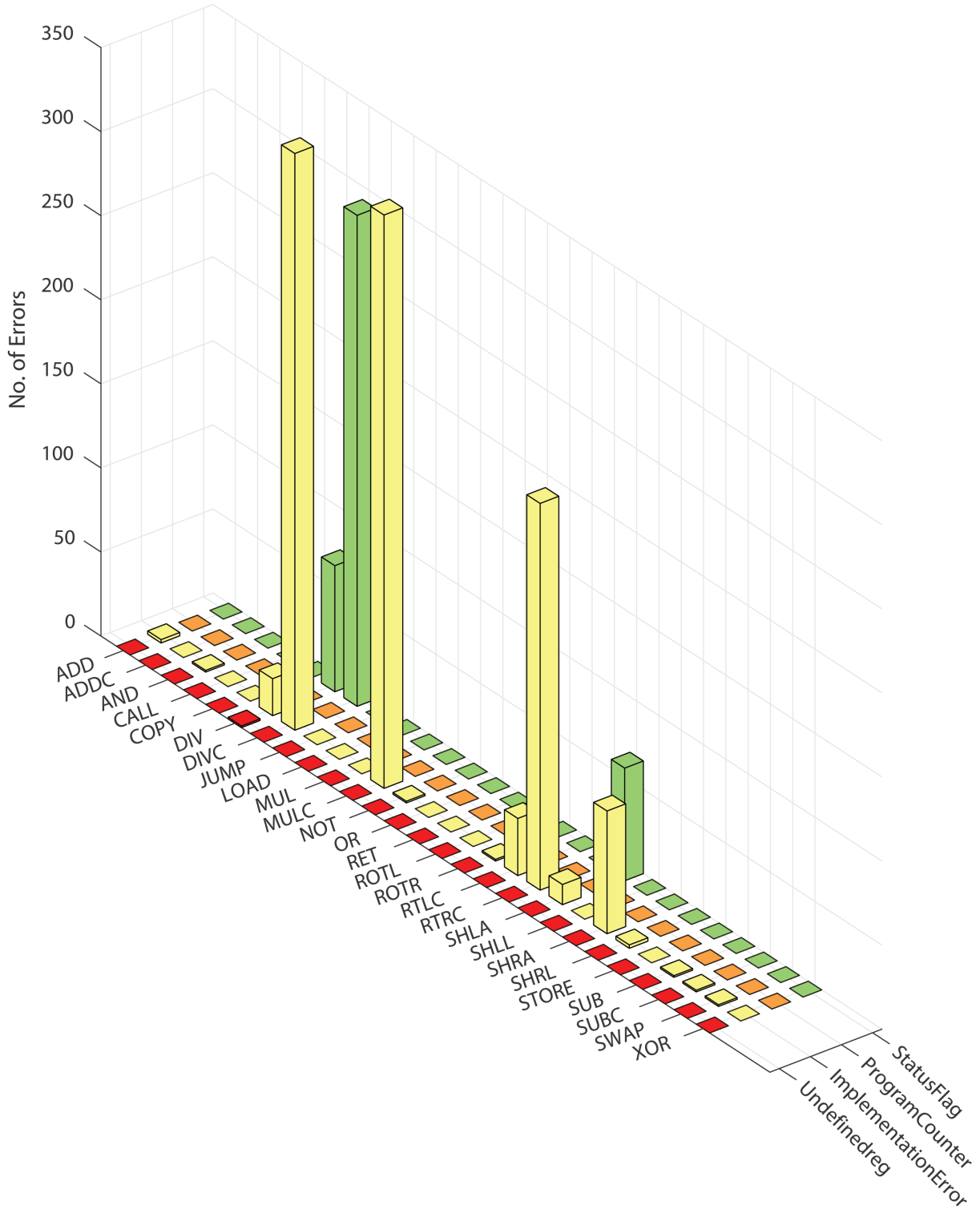

Figure 7.5: Errors for Test T1 - mode 'm'

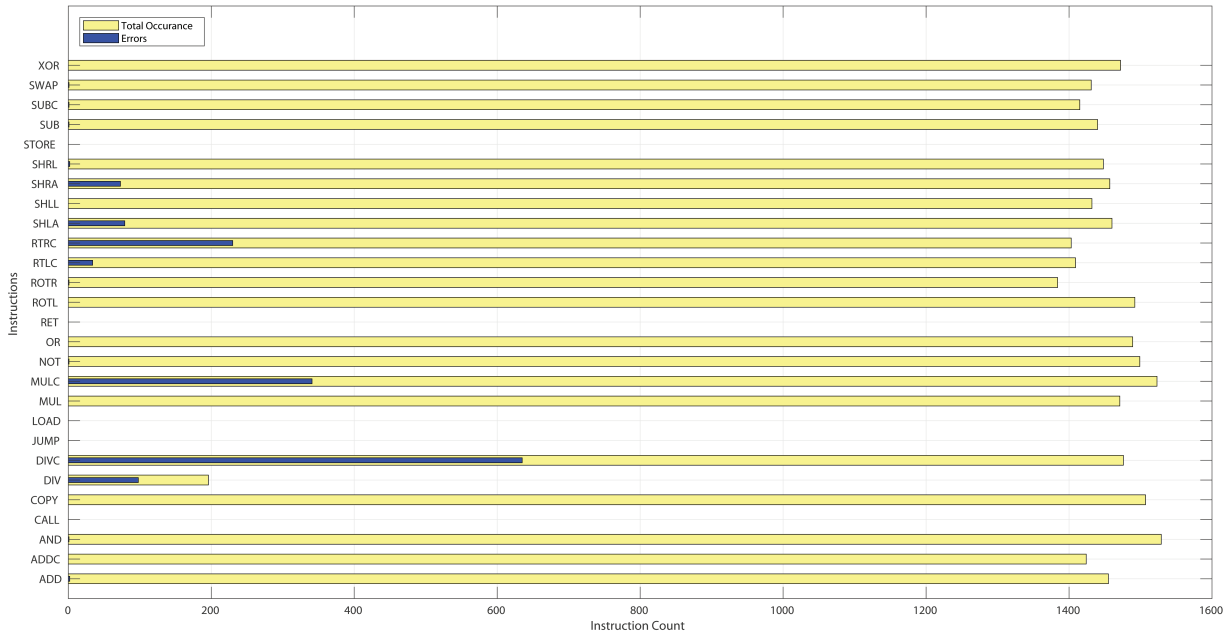

Figure 7.6: Total error count for Test T1 - mode 'm'

7.3.1 Mode 'mb' - Test T1

As seen in Figure 7.9 & 7.10 , 'CALL' and 'JUMP' have quite number of errors when compared to 'RET'. Errors related to 'RET' in the DUT are less as the instruction correctly popped the return address but had error in popping status register values. Hence, when the instruction following the 'CALL' didn't use status flag the error was masked.

7.3.2 Mode 'mb' - Test T2

In this test after correcting status flag in the DUT design, more errors are logged for the branch instruction than the previous test runs. This can be observed in Figure 7.11 & 7.12.

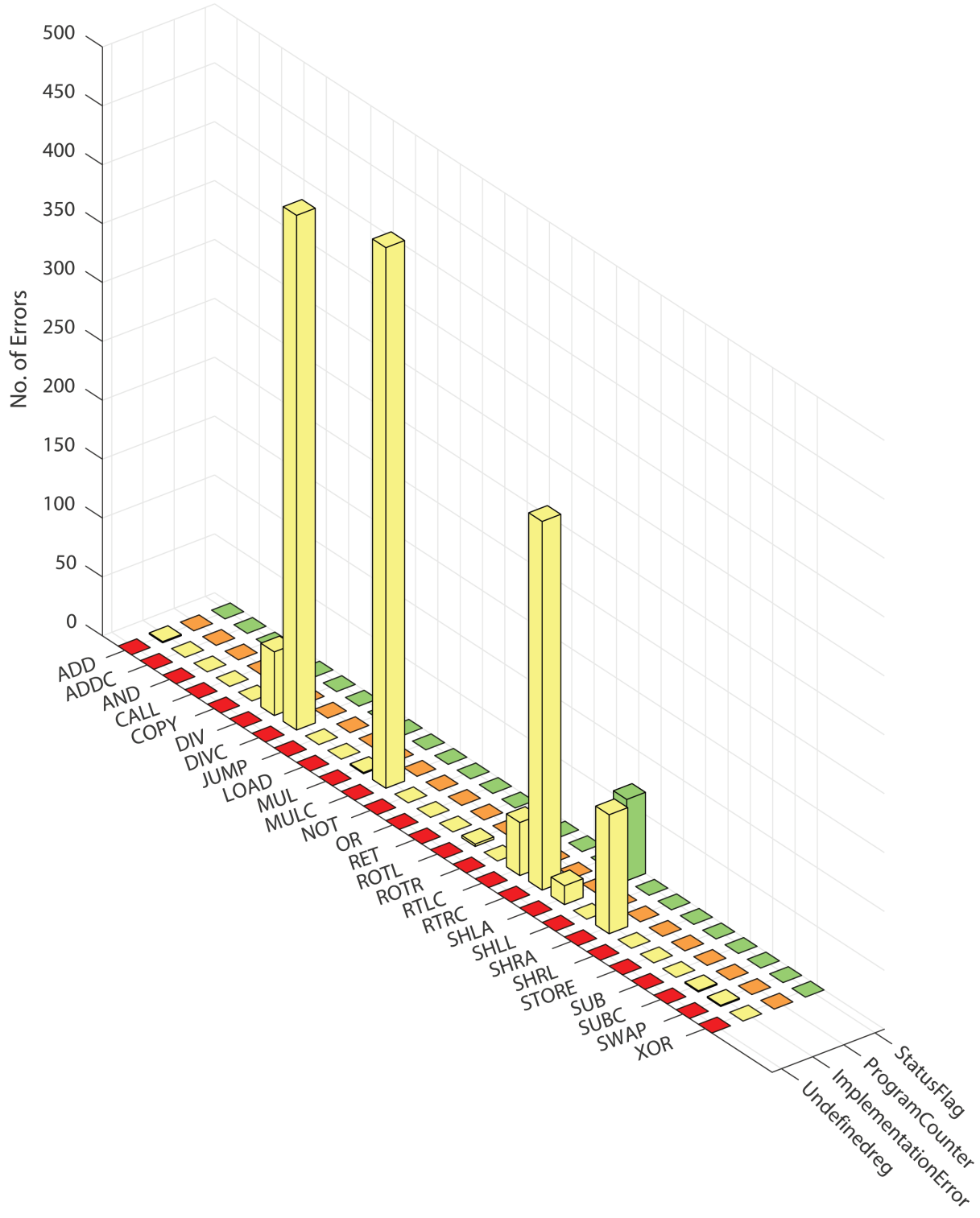

Figure 7.7: Errors for Test T2 - mode 'm'

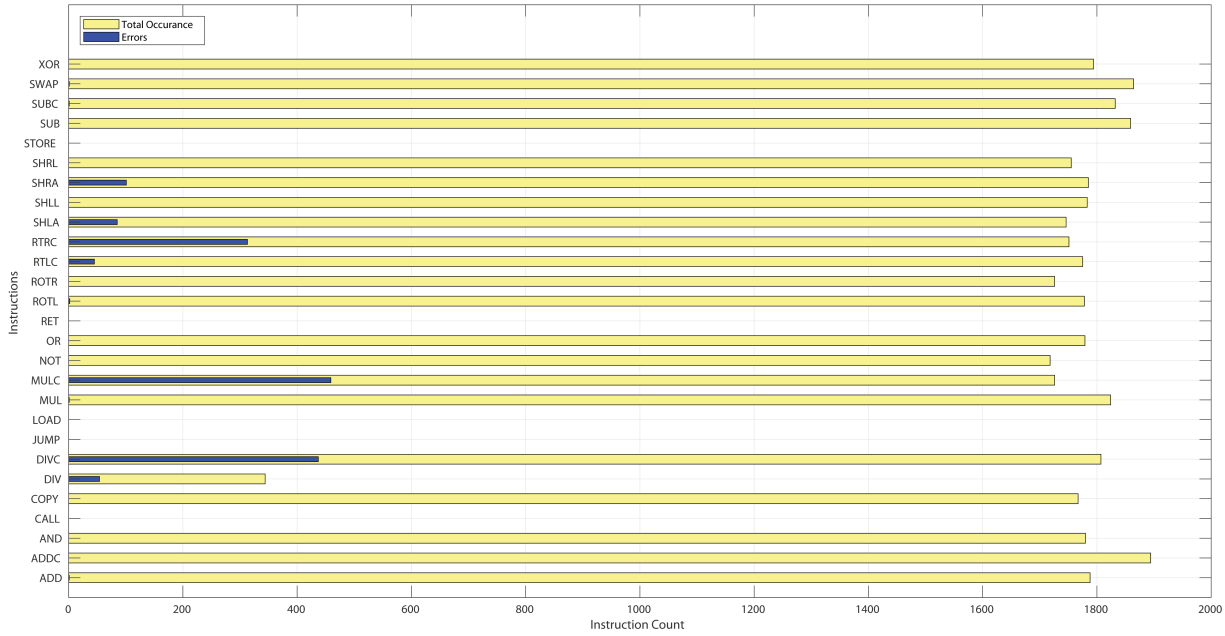

Figure 7.8: Total error count for Test T2 - mode ’m’

7.4 Mode ’md’

Manipulation and data transfer instructions are generated and tested in this mode. Errors in PC value suggest that the data transfer instruction calculated the memory address incorrectly. Other error for data transfer instruction is incorrect register value, whch can be due to incorrect value store at the address or memory read/write fault.

7.4.1 Mode ’md’ - Test T1

Figures 7.13 & 7.14 indicate a lot of errors related to both manipulation and data transfer instruction.

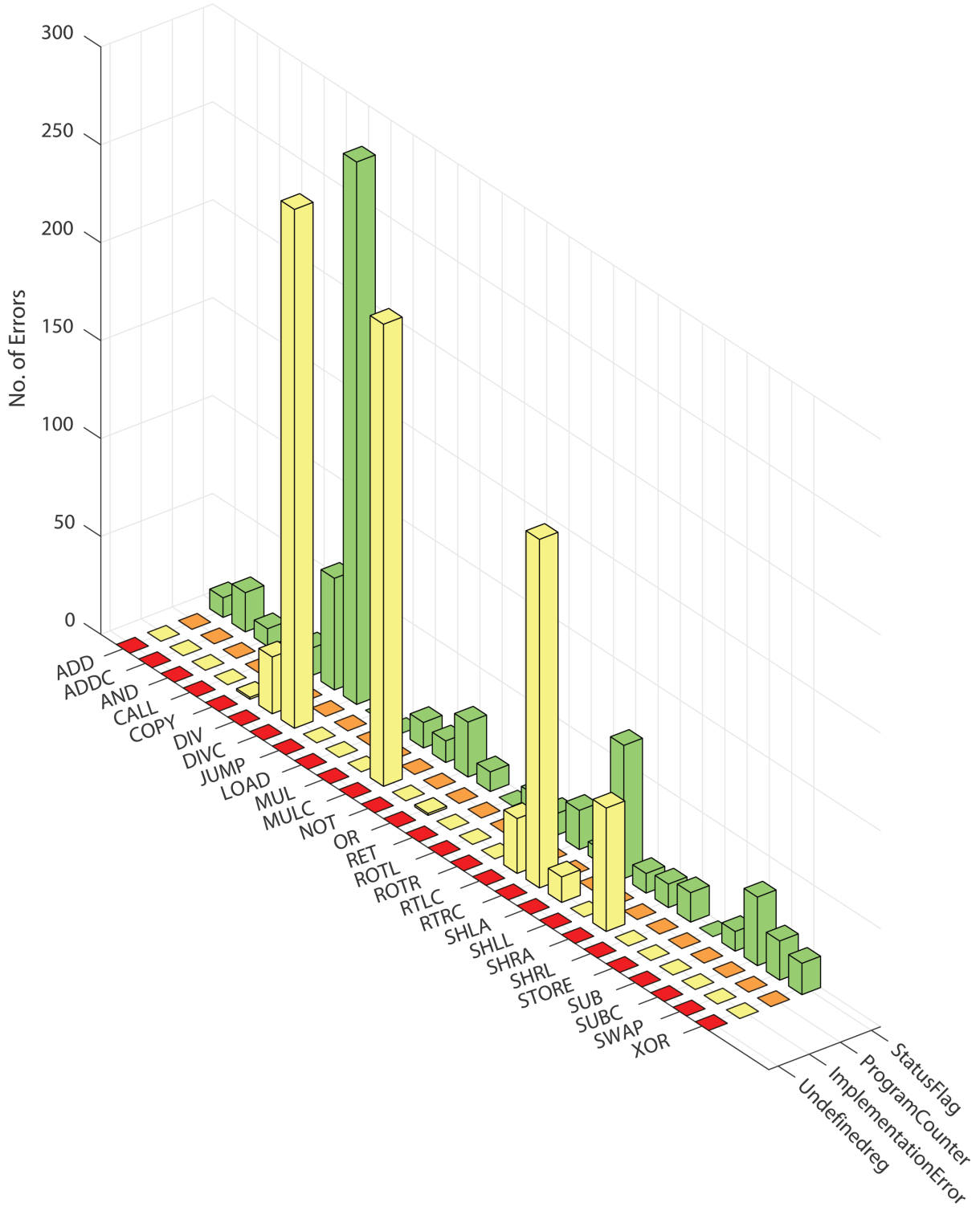

Figure 7.9: Errors for Test T1 - mode 'mb'

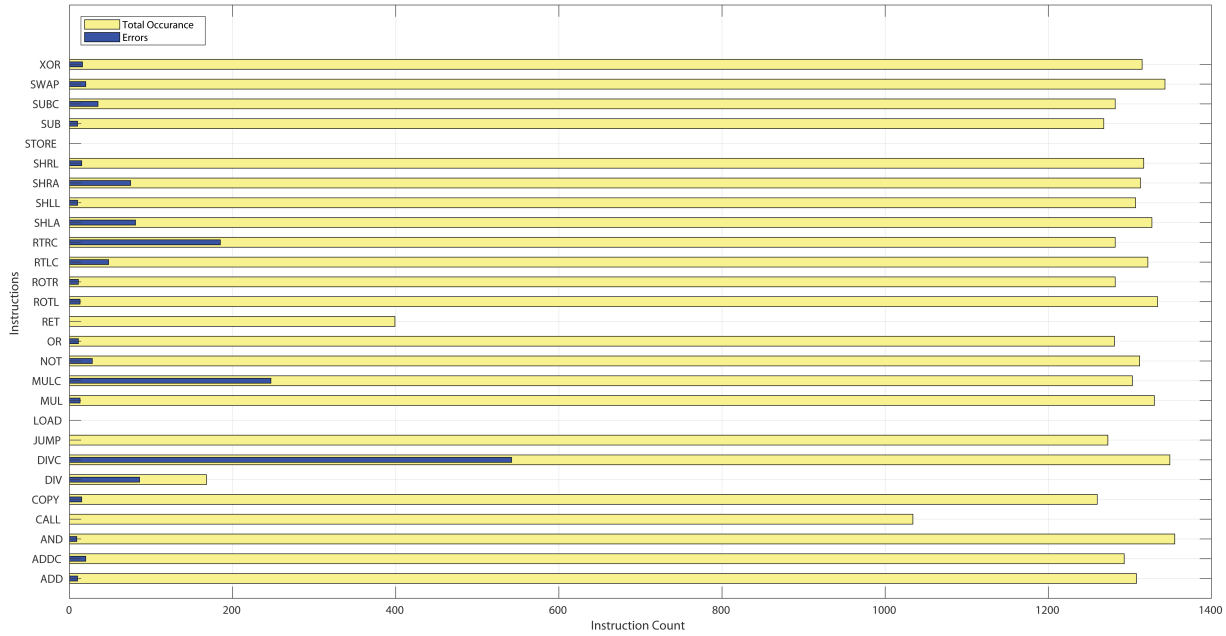

Figure 7.10: Total error count for Test T1 - mode ’mb’

7.4.2 Mode ’md’ - Test T2

After resolving errors related to status flags, Figures 7.15 & 7.16 in test T2 give a clearer picture that ’STORE’ along with some manipulation instructions (found in mode ’m’) have errors. With this information debugging can be started for the ’STORE’ instruction by the user.

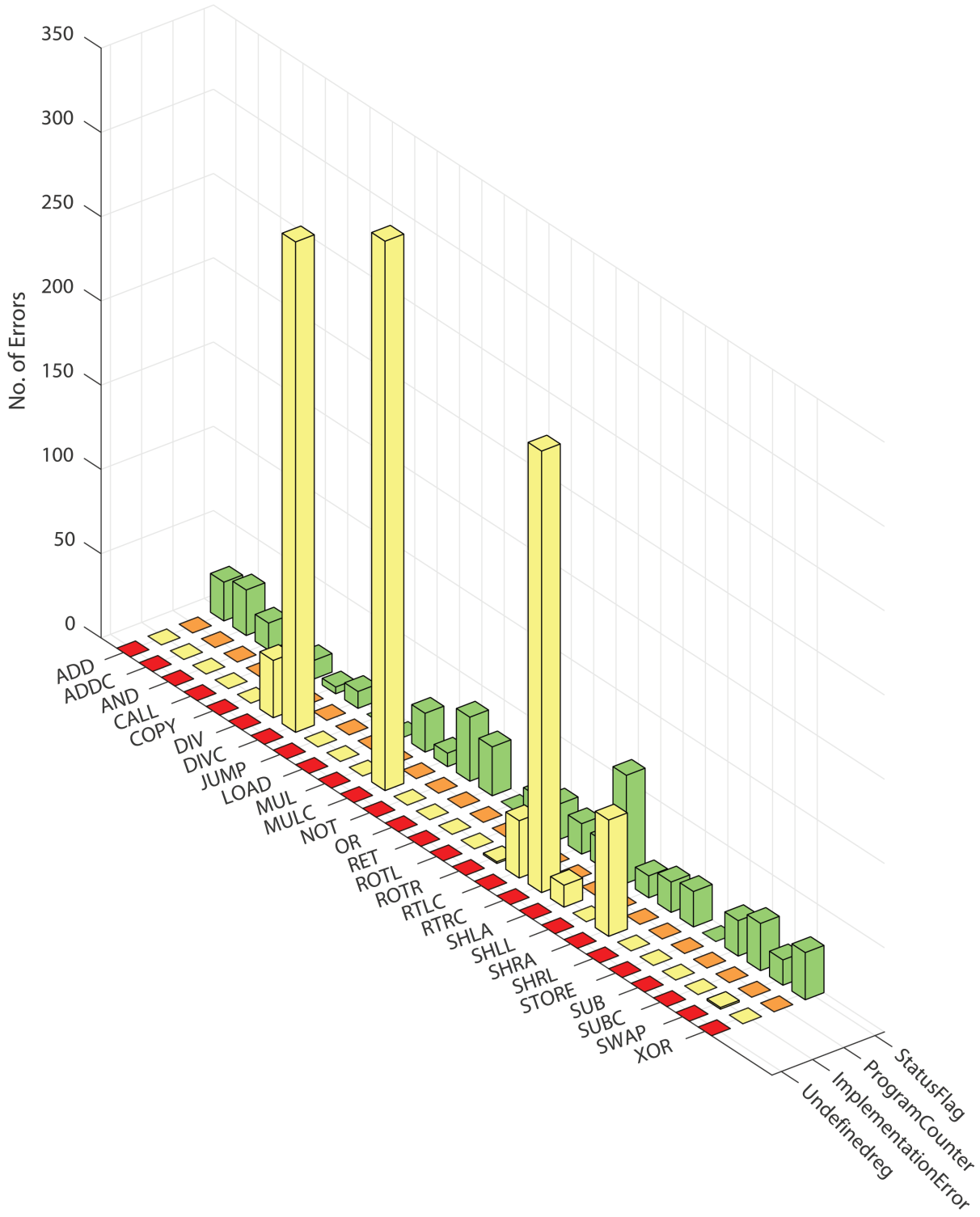

Figure 7.11: Errors for Test T2 - mode 'mb'

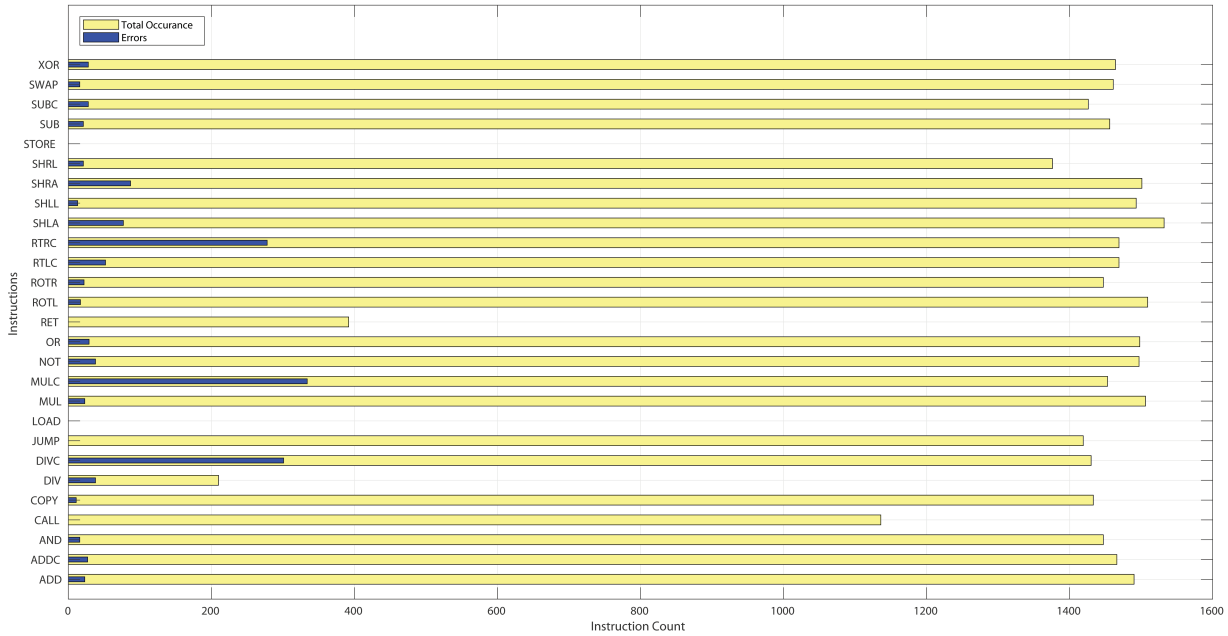

Figure 7.12: Total error count for Test T2 - mode ’mb’

Table 7.1: Tabulation of Errors for test *T1* (mode 'a') in processor kxmRISC621_v

Instructions	Total Occ	Errors	Ri	Rj	Status Flag	PC	Undef. Reg.	Implementation error
ADD	1892	10	3	1	9	0	0	1
ADDC	1886	25	4	0	25	0	0	0
AND	1880	10	0	0	10	0	0	0
CALL	1087	0	0	0	0	0	0	0
COPY	1898	12	0	0	12	0	0	0
DIV	690	100	12	0	75	0	0	25
DIVC	1917	493	45	0	246	0	0	247
JUMP	1904	0	0	0	0	0	0	0
LOAD	1703	0	0	0	0	0	0	0
MUL	1931	13	0	0	13	0	0	0
MULC	1847	274	49	0	8	0	0	266
NOT	1934	21	4	0	21	0	0	0
OR	1874	12	3	0	12	0	0	0
RET	502	0	0	0	0	0	0	0
ROTL	1945	21	1	0	21	0	0	0
ROTR	1899	17	2	0	16	0	0	1
RTLC	1939	45	8	0	12	0	0	33
RTRC	1883	180	19	0	6	0	0	174
SHLA	2027	88	5	0	59	0	0	29
SHLL	1882	17	2	0	16	0	0	1
SHRA	1899	73	15	0	11	0	0	62
SHRL	1887	14	1	0	14	0	0	0
STORE	1688	0	0	0	0	0	0	0
SUB	1981	17	1	0	17	0	0	0
SUBC	1881	30	4	0	30	0	0	0
SWAP	1954	12	3	0	12	0	0	0
XOR	1884	16	3	0	16	0	0	0

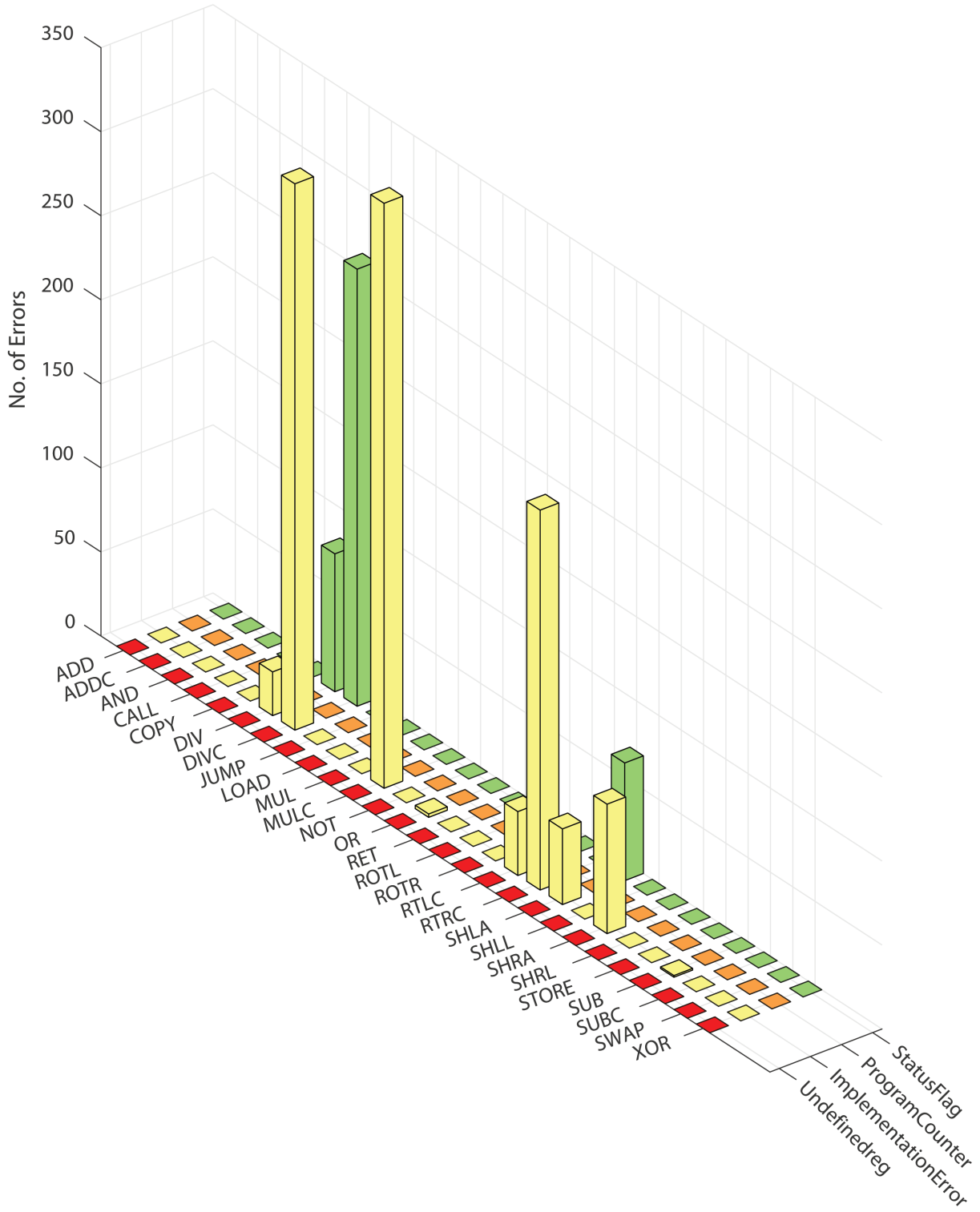

Figure 7.13: Errors for Test T1 - mode 'md'

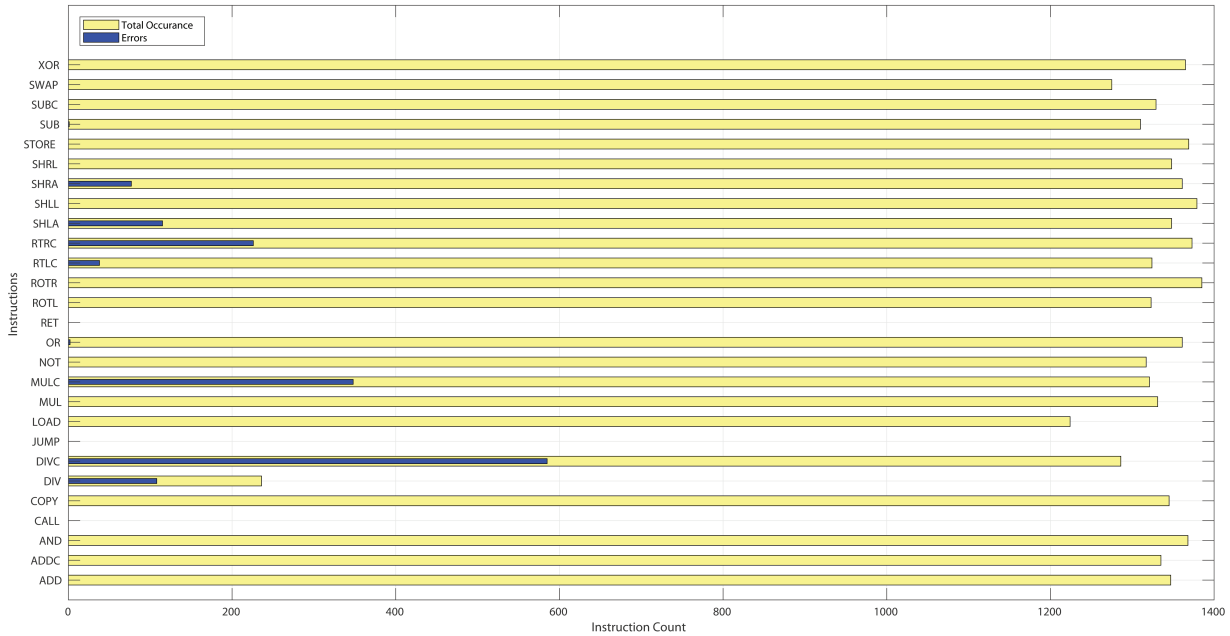

Figure 7.14: Total error count for Test T1 - mode ’md’

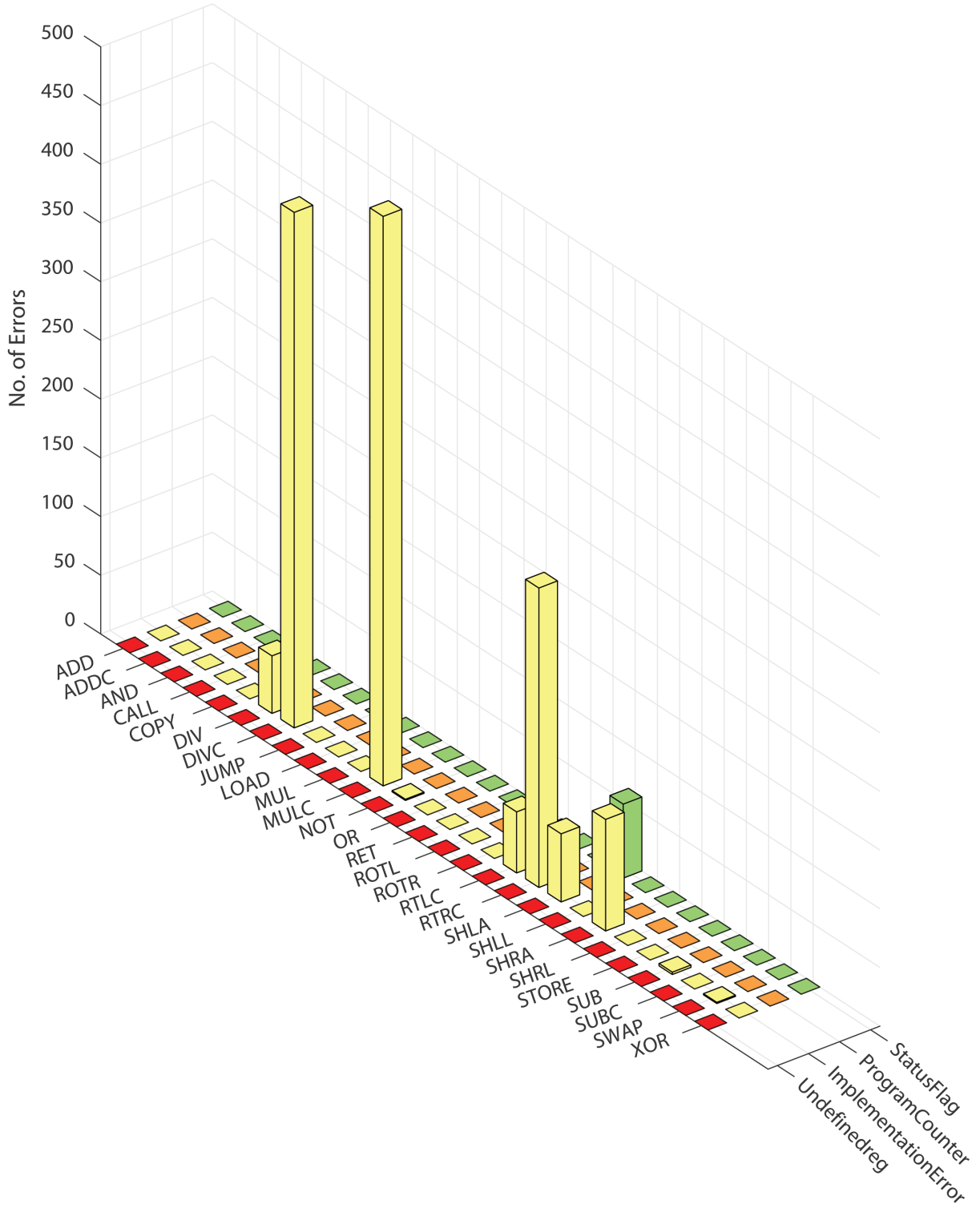

Figure 7.15: Errors for Test T2 - mode 'md'

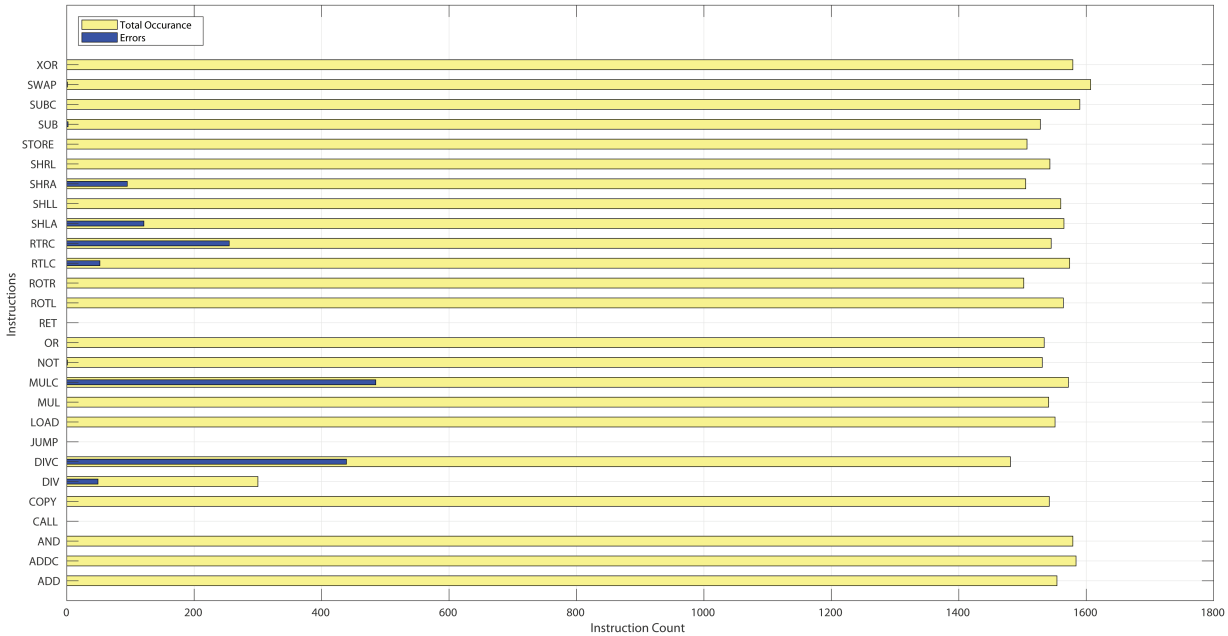

Figure 7.16: Total error count for Test T2 - mode ’md’

Chapter 8

Conclusion

In this work a Programmable Random Instruction Generator is developed. The instruction generator supports 12, 14 bit processor configurations for both Von Neumann and Harvard architecture. After running tests and looking at the gathered data two of the 8 samples are corrected to not have errors, this serves as proof of concept and helped development of the test environment. With a large number of tests run using various modes of instruction generation a very high coverage can be achieved. With flexible nature of the instruction generator, it can be easily adapted for higher bit processors e.g. 16 bits, 32 bits. The report database generator is based on instruction generator, mainly utilizing the file reading and information capture capabilities. Hence, report database generator also inherits the robustness of the instruction generator. The report and the graphs obtained after simulation are a powerful tool for debugging the design and architecture.

8.1 Future work

The current design of instruction generator can be further enhanced via following:

1. Option for users to select wider instruction word generation.

2. Support for floating point instructions.

3. Higher level in 'CALL' instruction nesting with 'JUMP' support inside subroutine.

4. Better presentation of result, the result should suggest corrective steps along side errors.

References

[1] D. Patru, *EEEE621 - Design of Computer Systems*. Rochester Institute of Technology, NY, USA, 2015.

[2] N. Pashupathy Manjula Devi, "Configurable verification of risc processors," Master's thesis, Rochester Institute of Technology, NY, USA, 5-2017. [Online]. Available: http://scholarworks.rit.edu/theses/9420

[3] D. Venkatesan and P. Nagarajan, "A case study of multiprocessor bugs found using ris generators and memory usage techniques," in *2014 15th International Microprocessor Test and Verification Workshop*, Dec 2014, pp. 4–9.

[4] C. A. Logan, "Directions in multiprocessor verification," in *Proceedings International Phoenix Conference on Computers and Communications*, Mar 1995, pp. 29–33.

[5] G. young Jeong, J. sung Park, H. woo Jo, B. woo Yoon, and M. jin Lee, "Arm7 compatible 32-bit risc processor design and verification," in *The 9th Russian-Korean International Symposium on Science and Technology*, 2005, pp. 607–610. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1507795&isnumber=32300

[6] T.-C. Chang, V. Iyengar, and E. M. Rudnick, "A biased random instruction generation envi-

ronment for architectural verification of pipelined processors," *Journal of Electronic Testing*, vol. 16, no. 1/2, pp. 13–27, 2000.

[7] S. Thiruvathodi and D. Yeggina, "A random instruction sequence generator for arm based systems," in *2014 15th International Microprocessor Test and Verification Workshop*, Dec 2014, pp. 73–77.

[8] M. Bose, J. Shin, E. M. Rudnick, T. Dukes, and M. Abadir, "A genetic approach to automatic bias generation for biased random instruction generation," in *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No.01TH8546)*, vol. 1. IEEE, May 2001, pp. 442–448 vol. 1.

[9] M. Bose, E. M. Rudnick, and M. Abadir, "Automatic bias generation using pipeline instruction state coverage for biased random instruction generation," in *Proceedings Seventh International On-Line Testing Workshop*, 2001, pp. 65–71.

[10] H. Iwashita, S. Kowatari, T. Nakata, and F. Hirose, "Automatic program generator for simulation-based processor verification," in *Proceedings of IEEE 3rd Asian Test Symposium (ATS)*, Nov 1994, pp. 298–303.

[11] H. P. Klug, "Microprocessor testing by instruction sequences derived from random patterns," in *International Test Conference*, Sep. 1988, pp. 73–80. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=207782&isnumber=5318

[12] J. Rault, "A graph theoretical and probabilistic approach to the fault detection of digital circuits," *INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, DIGEST OF PAPERS, PASADENA, CALIF, Mar 1971*, pp. 26–29, Mar 1971.

[13] J. Hudson and G. Kurucheti, "A configurable random instruction sequence (ris) tool for

memory coherence in multi-processor systems," in *2014 15th International Microprocessor Test and Verification Workshop*, Dec 2014, pp. 98–101.

[14] S. P. Jung, J. Xu, D. Lee, J. S. Park, K. joo Kim, and K. shik Cho', "Design & verification of 16 bit risc processor," in *2008 International SoC Design Conference*, vol. 03, Nov 2008, pp. III–13–III–14.

[15] B. Amal and B. Sur, *SystemC and SystemC-AMS in Practice*. Springer Verlag, 2014.

[16] J. Stoppe and R. Drechsler, "Analyzing systemc designs: Systemc analysis approaches for varying applications," *Sensors*, vol. 15, no. 5, pp. 10 399–10 421, 2015. [Online]. Available: http://www.mdpi.com/1424-8220/15/5/10399

[17] M. Y. Vardi, "Formal techniques for systemc verification; position paper," in *2007 44th ACM/IEEE Design Automation Conference*, June 2007, pp. 188–192.

[18] F. Bruschi, F. Ferrandi, and D. Sciuto, "A framework for the functional verification of systemc models," *International Journal of Parallel Programming*, vol. 33, no. 6, p. 667, Dec 2005. [Online]. Available: https://doi.org/10.1007/s10766-005-8908-x

[19] S. Vijayaraghavan and M. Ramanathan, *A Practical Guide for SystemVerilog Assertions*, 1st ed. Springer US, 2005.

[20] L. Chai, Z. Xie, and X. Wang, "A verification methodology for reusable test cases and coverage based on system verilog," in *2014 IEEE International Conference on Electron Devices and Solid-State Circuits*, June 2014, pp. 1–2.

[21] M. Keaveney, A. McMahon, N. O'Keeffe, K. Keane, and J. O'Reilly, "The development of advanced verification environments using system verilog," in *IET Irish Signals and Systems Conference (ISSC 2008)*, June 2008, pp. 325–330.

[22] S. R, J. S, R. A. Rahiman, R. Karthik, A. M. S, and S. S. S, "Verification of a risc processor ip core using systemverilog," in *2016 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*, March 2016, pp. 1490–1493.

[23] T. P. F. d. S. e. Lima, "Reconfigurable model for risc processors," Master's thesis, Rochester Institute of Technology, NY, USA, 12-2016. [Online]. Available: http://scholarworks.rit.edu/theses/9326

[24] C. Spear and G. Tumbush, *SystemVerilog for Verification, Third Edition: A Guide to Learning the Testbench Language Features*. Springer Publishing Company, Incorporated, 2012.

[25] S. Sutherland, S. Davidmann, and P. Flake, *SystemVerilog for Design Second Edition*, 2nd ed. Springer US, 2006.

[26] P. Hammarlund, A. J. Martinez, A. A. Bajwa, D. L. Hill, E. Hallnor, H. Jiang, M. Dixon, M. Derr, M. Hunsaker, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, S. Gunther, T. Piazza, and T. Burton, "Haswell: The fourth-generation intel core processor," *IEEE Micro*, vol. 34, no. 2, pp. 6–20, Mar 2014.

[27] P. M. Kogge, *Architecture of Pipelined Computers*, 1st ed. CRC Press, Jan. 1981.

[28] T. Christiansen, J. Orwant, L. Wall, and B. foy, *Programming Perl*, 4th ed. O'Reilly Media, Mar. 2012.

Appendix I

RISC Processor Instructions

I.1 Manipulation Instructions

Table I.1: Manipulation Instructions[1, 2]

Instruction	**Description**	**Operation**	**Syntax**	**Example**	**Machine Code for 12 bits**
ADD	Addition of two registers	Ri = Rj + Ri	ADD <Ri>, <Rj>;	ADD R6, R7;	<6-bit opcode for ADD> 110 111
ADDC	Addition of a register and a constant	Ri = Ri + CON-STANT at Rj	ADDC <Ri>, <Constant at Rj field>;	ADDC R2, 1;	<6-bit opcode for ADDC> 010 001

SUB	Subtraction of two registers	Ri = Ri - Rj	SUB <Ri>, <Rj>;	SUB R6, R7	<6-bit opcode for SUB> 110 111
SUBC	Subtraction of a register and a constant	Ri = Ri - CONSTANT at Rj	SUBC <Ri>, <Constant at Rj field>;	SUBC R2, 1;	<6-bit opcode for SUBC> 010 001
MUL	Multiplication of two registers	Ri = Rj * Ri	MUL <Ri>, <Rj>;	MUL R6, R7;	<6-bit opcode for MUL> 110 111
MULC	Multiplication of a register by a constant	Ri = Ri * CONSTANT at Rj	MULC <Ri>, <Constant at Rj field>;	MULC R2, 3;	<6-bit opcode for MULC> 010 011
DIV	Division of a register by another register	Ri = Ri / Rj	DIV <Ri>, <Rj>;	DIV R6, R7;	<opcode for DIV> 110 111
DIVC	Division of a register by a constant	Ri = Ri / CONSTANT at Rj	DIVC <Ri>, <Constant at Rj field>;	DIVC R2, 3	<code for DIVC> 010 011

NOT	Logical Negation of a register	Ri = NOT Ri	NOT <Ri>;	NOT R7;	<code for NOT> 111 XXX
XOR	Logical XOR (Exclusive OR) of two registers	Ri = Ri XOR Rj	XOR <Ri>, <Rj>;	XOR R7, R6;	<code for XOR> 111 110
SHLL	Shift Left Logic the value of a register by a constant	Ri = Ri << Constant	SHLL <Ri>, <Constant at Rj field>;	SHLL R7, 0x2;	<code for SHLL> 111 010
SHRL	Shift Right Logic the value of a register by a constant	Ri = Ri >> Constant	SHRL <Ri>, <Constant at Rj field>;	SHRL R1, 0x5;	<code for SHRL> 001 101
SHLA	Shift Left Arithmetic the value of a register by a constant.	Ri = Ri << Constant	SHLA <Ri>, <Constant at Rj field>;	SHLA R2, 0x4;	<code for SHRL> 010 100

SHRA	Shift Right Arithmetic the value of a register by a constant.	Ri = Ri >> Constant	SHRA <Ri>, <Constant at Rj field>;	SHRA R3, 0x4;	<code for SHRA> 011 100
ROTL	Rotate Left the value of a register by a constant	Ri = Ri ROTL Constant	ROTL <Ri>, <Constant at Rj field>;	ROTL R2, 0x4;	<code for ROTL> 010 100
ROTR	Rotate Right the value of a register by a constant	Ri = Ri ROTR Constant	ROTR <Ri>, <Constant at Rj field>;	ROTR R6, 0x2;	<code for ROTR> 110 010
RTLC	Rotate Left the value of a register through carry by a constant	Ri = Ri RTLC Constant	RTLC <Ri>, <Constant at Rj field>;	RTLC R2, 0x1;	<code for RTLC> 010 001
RTRC	Rotate Right the value of a register through carry by a constant	Ri = Ri RTRC Constant	RTRC <Ri>, <Constant at Rj field>;	RTRC R6, 0x7;	<code for RTRC> 110 111

SWAP	Swap the data of two registers	Ri = Rj and Rj=Ri	SWAP <Ri>, <Rj>;	SWAP R2, R3;	<code for SWAP> 010 011
COPY	Copy data from register into another	Ri receives Rj value;	CPY <Ri>, <Rj>;	CPY R5, R7;	<code for CPY> 101 111

I.2 Data Transfer Instructions

Table I.2: Transfer Instructions[1, 2]

Instruction	**Description**	**Operation**	**Syntax**	**Example**	**Machine Code for 12 bits**
STORE - Direct addressing mode	The value of the register is written to a memory address specified by the instruction	Memory Address = R[Rj]	STORE R0, M[<Rj>, Address];	STORE R0, M[R4, 0x800];	Inst 1: <code for STORE> 000 010 Inst 2: 0100 0000 0000
STORE - PC addressing mode	The value of the register is stored at the memory address, calculated as sum of offset address and the PC value	Offset Address + PC = R[Rj]	STORE R1, M[<Rj>, Address];	STORE R1, M[R2, 0x405];	Inst 1: <code for STORE> 001 010 Inst 2: 0100 0000 0101

STORE - Stack Pointer	The value of the register is stored at the memory address, pointed by the Stack Pointer	Memory Address referent to SP value = R[Rj]	STORE R2, M[<Rj>, Address];	STORE R2, M[R3, 0x700];	Inst 1: <code for STORE> 010 101 Inst 2: 0100 0000 0111
STORE - Register addressing mode	The value of the register is stored at the memory address, calculated as sum of offset address and the register value	Offset Address + R[Ri] = R[Rj]	STORE <Ri>, M[<Rj>, Address]; [1]	ST R3, M[R2, 0x40F];	Inst 1: <code for STORE> 011 010 Inst 2: 0100 0000 1111
LOAD - Direct address mode	The value of a memory address specified is loaded in the register	R[Rj] = Data at the offset address	LOAD R0, M[<Rj>, Addr];	LOAD R0, M[R1, 0x860];	Inst 1: <code for LOAD> 000 001 Inst 2: 0100 0000 0000

LOAD - PC addressing mode	The value of a memory address, calculated as sum of offset address and the PC value, is loaded in the register	R[Rj] = Data at the (offset address + PC)	LOAD R1, M[<Rj>, Address];	LOAD R1, M[R7, 0x204];	Inst 1: <code for LOAD> 001 111 Inst 2: 0100 0000 0010
LOAD - Stack Pointer	The value of a memory address, pointed by the Stack Pointer is loaded in the register	R[Rj] = Memory Address referent to Stack Pointer value	LOAD R2, M[<Rj>, Address];	LOAD R2, M[R9, 0x600];	Inst 1: <code for LOAD> 001 110 Inst 2: 0100 0000 0011

LOAD - Register addressing mode	The value of a memory address, calculated as sum of offset address and the register value is loaded in the register	R[Rj] = Data of the (Offset address + R[Ri])	LD <Ri>, M[<Rj>, Address]; [2]	LOAD R3, M[R7, 0x4FF];	Inst 1: <code for LOAD> 011 111 Inst 2: 0100 1111 1111

[1] Ri represents registers from R3 to R7 for 12-bit processor. For 14-bit processor, from registers R3 to R15.

[2] Ri represents registers from R3 to R7 for 12-bit processor, registers R3 to R15 for 14-bit processor.

I.3 Branch Instructions

Table I.3: Branch Instructions[1, 2]

Instruction	**Description**	**Operation**	**Syntax**	**Example**	**Machine Code for 12 bits**

JMPU - Jump Unconditional	Jump to a memory location. This location is determined by the current value of Ri. [3]	PC = New Address;	Code line 1: JMPU @jmp_1 ... Code line 2: @jmp_1 <MNEMONICS>	Code line 1: JMPU @jmp_2; ... Code line 2: @jmp_2 SUBC R0, 0x1;	Inst 1: <code for JMPU> <xx><0000> Inst 2: 0000 0011 1111 [4]
JMPC - Jump if Carry	Jump to a memory location , if carry status bit is set. [3]	PC = New Address;	Code line 1: JMPC @jmp_1 ... Code line 2: @jmp_1 <MNEM>	Code line 1: JMPC @jmp_2; ... Code line 2: @jmp_2 SUBC R5, 0x2;	Inst 1: <code for JMPC> <xx><1000> Inst 2: 0000 0000 1111 [5]
JMPN - Jump if Negative	Jump to a memory location , if Negative status bit is set. [3]	PC = New Address;	Code line 1: JMPN @jmp_label ... Code line 2: @jmp_label <MNEM>	Code line 1: JMPN @jmp_c; ... Code line 2: @jmp_c SUBC R4, 0x3;	Inst 1: <code for JMPN> <xx><0100> Inst 2: 0000 0000 1111 [6]

JMPV - Jump if Overflow	Jump to a memory location , if Overflow status bit is set. [3]	PC = New Address;	Code line 1: JMPV @jmp_label ... Code line 2: @jmp_label <MNEM>	Code line 1: JMPV @jmp_v; ... Code line 2: @jmp_v SUBC R5, 0x5;	Inst 1: <code for JMPV> <xx><0010> Inst 2: 0000 0000 1111 [7]
JMPZ - Jump if Zero	Jump to a memory location , if Zero status bit is set. [3]	PC = New Address;	Code line 1: JMPZ @jmp_label ... Code line 2: @jmp_label <MNEM>	Code line 1: JMPZ @jmp_z; ... Code line 2: @jmp_z DIVC R1, 0x2;	Inst 1: <code for JMPZ> <xx><0001> Inst 2: 0000 0000 1111 [8]

JMPNC - Jump if NOT Carry	Jump to a memory location , if Carry status bit is zero and overflow, zero and negative status bits are set as ’1’. [3]	PC = New Address;	Code line 1: JMPNC @jmp_label ... Code line 2: @jmp_label <MNEM>	Code line 1: JMPNC @jmp_nc; ... Code line 2: @jmp_nc SUBC R4, 0x3;	Inst 1: <code for JMPNC> <xx><0111> Inst 2: 0000 0000 1111 [9]
JMPNN - Jump if NOT Negative	Jump to a memory location , if Negative status bit is zero and carry, overflow and negative status bits are set as ’1’. [3]	PC = New Address;	Code line 1: JMPNN @jmp_label ... Code line 2: @jmp_label <MNEM>	Code line 1: JMPNN @jmp_nc; ... Code line 2: @jmp_nn DIVC R3, 0x2;	Inst 1: <code for JMPNN> <xx><1011> Inst 2: 0000 0000 1111 [10]

JMPNV - Jump if NOT Overflow	Jump to a memory location , if Overflow status bit is zero and Carry, Zero and Negative status bits are set as ’1’. [3]	PC = New Address;	Code line 1: JMPNV @jmp_label ... Code line 2: @jmp_label <MNEM>	Code line 1: JMPNV @jmp_nc; ... Code line 2: @jmp_nv MULC R3, 0xA;	Inst 1: <code for JMPNV> <xx><1101> Inst 2: 0000 0000 1111 [11]
JMPNZ - Jump if NOT Zero	Jump to a memory location , if Zero status bit is zero and Carry, Overflow and Negative status bits are set as ’1’. [3]	PC = New Address;	Code line 1: JMPNZ @jmp_label ... Code line 2: @jmp_label <MNEM>	Code line 1: JMPNZ @jmp_nc; ... Code line 2: @jmp_nn MULC R2, 0x9;	Inst 1: <code for JMPNZ> <xx><1110> Inst 2: 0000 0000 1111 [12]

Call	Call instruction branches to a specific address and the current address & status flags are stored on the stack	PC = Address1 -> Address2	call @label <MNEM>	**Code Line 1:** CALL @name; ... **Code Line 2:** @name MULC R1, 0x1;	Inst 1: <code for CALL> xxx xxx Inst 2: 0000 1111 1111 [13]
RET	Returns back to the instruction after CALL instruction. It does so by restoring the PC value and status flags from values stored on the stack by Call instruction	Address <- RET	RET	RET	<code for RET> xxx xxx

[3]if Ri = 0, direct addressing.

if Ri = 1, PC addressing.

if Ri = 2, Stack Pointer addressing.

if Ri greater than 2, register addressing.

[4]JMPU: Two machine codes are present for JMPU, one machine code specific for the instruction code, Jump unconditional, and other machine code that represents the destination address. Rj must be zero.

Obs:The instruction that Jump unconditional is pointing is at address 0x03F and all bits relative to CNVZ status (Carry, Negative, Overflow and Zero flags) are zero. According to the value of Ri, Program Counter value will change.

[5]JMPC: Two machine codes are present for JMPC, one machine code for the instruction code, Jump if Carry, and other machine code specific for the destination address. The status bits must be 4'b1000, or 0x8 for 12 and 14-bit processors. For 16-bit processor the status bits must be 4'b10000 or 0x10 .

Obs:The instruction that Jump if Carry is pointing is at address 0x00F and all bits relative to CNVZ status is 0x8. According to the value of Ri, Program Counter value will change.

[6]JMPN: Two machine codes are present here, one machine code for the instruction code, Jump if Negative, and other for the destination address. The status bits must be 4'b0100, or 0x4 for 12 and 14 bits and 4'b01000, or 0x8 for 16 bits.

Obs:The instruction that Jump if Negative is pointing is at address 0x00F and all bits relative to CNVZ status are 0x4. According to the value of Ri, PC value will change.

[7]JMPV: Two machine codes are present here for this instruction code, one machine code for the instruction code, Jump if Overflow, and other for the destination address. The status bits must be 4'b0010, or 0x2 for 12 and 14 bits and 4'b0100, or 0x4 for 16 bits.

Obs:The instruction that Jump if Overflow is pointing is at address 0x00F and all bits relative to CNVZ status are 0x2. According to the value of Ri, Program Counter value will change.

[8]JMPZ: Two machine codes are present for this instruction, one machine code for the instruction code, Jump if Zero, and other for the destination address. The status bits must be 4'b0001, or 0x1 for 12 and 14 bits and 4'b0010, or 0x2 for 16 bits.

Obs:The instruction that Jump if Zero is pointing is at address 0x00F and all bits relative to CNVZ status are 0x1. According to the value of Ri, Program Counter value will change.

[9]JMPNC: Two machine codes are present for this instruction, one machine code for the instruction code, Jump if Not Carry, and other for the destination address. The status bits must be 4'b0111, or 0x7 for 12 and 14-bit processors. For 16-bit processor the status bits must be 5'b01110 or 0xE .

Obs:The instruction that Jump if Carry is pointing is at address 0x00F and all bits relative to CNVZ status is 0x7. According to the value of Ri, Program Counter value will change.

[10]JMPNN: Two machine codes are present for this instruction, one machine code for the instruction code, Jump if Not Negative and other for the destination address. The status bits must be 4'b1011, or 0xB for 12 and 14-bit processors. For 16-bit processor the status bits must be 5'b10110 or 0x16.

Obs:The second instruction represents the offset address for Jump if Not Negative. It is pointing is at address 0x00F and all bits relative to CNVZ status are 0xB. According to the value of

Ri, Program Counter value will change.

[11]JMPNV: Two machine codes are present here, one machine code for the instruction code, Jump if Not Overflow and other for the destination address. The status bits must be 4'b1101, or 0xD for 12 and 14-bit processors. For 16-bit processor the status bits must be 5'b11010 or 0x1A.

Obs:The second instruction represents the offset address for Jump if Not Overflow. It is pointing is at address 0x00F and all bits relative to CNVZ status are 0xD. According to the value of Ri, Program Counter value will change.

[12]JMPNZ: Two machine codes are present, one machine code for the instruction code, Jump if Not Zero and other for the destination address. The status bits must be 4'b1110, or 0xE for 12 and 14-bit processors. For 16-bit processor the status bits must be 5'b11100 or 0x1C.

Obs:The second instruction represents the offset address for Jump if Not Zero. It is pointing is at address 0x00F and all bits relative to CNVZ status are 0xE. According to the value of Ri, PC value will change.

[13]Call: Two machine codes are present, one for the instruction code CALL and one for the next destination address.

Obs: The instruction that Call is pointing is at address 0x0FF.

Appendix II

Configuration File

```
# Configuration File
   #
   #name of the folder where the processor files are located
   name_folder:  2_kxm
   #name of the processor in format <filename.ext> with the extension
   name:  kxmRISC621_v.v
   #number of bits for the processor
   bits:0x0C
   #number of registers
   registers:0x8
   #Harvard(0) or Von Neumann (1)
   architecture:1
   #Opcode size
   opcode_size:0x6
   #define the operand sizes for manipulation instructions, LOAD, STORE, COPY
```

```
and SWAP
   #for JUMP, CALL and RET, Ri will be defined by:(number of bits) - (4+opcode_size)
   operand1_size:0x3
   operand2_size:0x3
   #operand1_size + operand2_size + opcode_size has to be equal to bits
   #data memory address bus size
   dm_size:0xA
   #Program and Data Memory Size VN in power of 2
   memory_size:0x0C
   #Value of PC used in calculating effective address:  '0' if current instruction's
PC is used, '1' if next instruction's PC is used
   pc_in_pc_relative:0
   #Stack Pointer top
   SP: 0x1FEF
   #Stack growth direction up(i.e.lower to higher address "0") or down (i.e.higher
to lower address "1")
   Stack_direction:  1
   #Percentage of memory reserved for stack
   Stack_size:  2
   #opcodes
   #instruction mapping:  the left hand colum represent user mnemonics and right
hand column are reserved mnemonics (with their explaination) reserved by the
test environment
   #NOTE: mapping should occur before mnemonic :  opcode definition
   start_mapping:
```

```
ADD : ADD ;

SUB : SUB ;

ADDC : ADDC ;

SUBC : SUBC ;

MUL : MUL ;

DIV : DIV ;

MULC : MULC ;

DIVC : DIVC ;

NOT : NOT ;

AND : AND ;

OR : OR ;

XOR : XOR ;

SHLL : SHLL ;

SHRL : SHRL ;

SHLA : SHLA ;

SHRA : SHRA ;

ROTL : ROTL ;

ROTR : ROTR ;

RTLC : RTLC ;

RTRC : RTRC ;

COPY : COPY ;

SWAP : SWAP ;

LOAD : LOAD ;

STORE : STORE ;

JUMP : JUMP ;
```

```
CALL : CALL ;

RET : RET ;

end_mapping:

opcode:

start_data_transfer:

LOAD :0x00 ;

STORE :0x01 ;

end_data_transfer:

start_manipulation:

ADD : 0x02 ;

SUB : 0x03 ;

ADDC : 0x04 ;

SUBC : 0x05 ;

NOT : 0x06 ;

AND : 0x07 ;

OR : 0x08 ;

XOR : 0x09 ;

SHLL : 0x0A ;

SHRL : 0x0B ;

SHLA : 0x0C ;

SHRA : 0x0D ;

ROTL : 0x0E ;

ROTR : 0x0F ;

RTLC : 0x10 ;

RTRC : 0x11 ;
```

```
MUL : 0x12 ;
DIV : 0x13 ;
MULC : 0x14 ;
DIVC : 0x15 ;
COPY : 0x16 ;
SWAP : 0x17 ;
end_manipulation:
start_branch:
JUMP : 0x18 ;
CALL : 0x21 ;
RET : 0x22 ;
end_branch:
#clock cycle at which the first instruction output is obtained
clk_st:5
#name of the files used inside processor in format <filename.ext>
#note:  use only 'name_pm' for von neumann arch
name_pm:  kxmRISC621_ram1.v
name_dm:
name_div:  kxmRISC_div.v
name_mul:  kxmRISC_mult.v
name_cnt:  kxm621_count.v
#are you for stalling ld, st, jmp, cal or ret?
#i.e., are you delaying the fetch of next instruction?  '1'-yes, '0'-no
del_ld:1
del_st:1
```

```
del_jmp:1

#del_cal:1

#del_ret:1

EOF:
```

Appendix III

Source Code

III.1 Random Instruction Generator

```
use Getopt::Long;
use strict;
use warnings;
use List::Util 'shuffle';
## sf as ZVNC
##declaration

my %user_input = @ARGV;
my $config = "";
my $mode = 'a';
our $num_reg = 0; ## number of registers
our $bus_size = 0;
our $arch = 0; ## 0 = harvard; 1= Von Neumann
our $opcode_size = 0;
our $op1_size = 0;
our $op2_size = 0;
my $memory_size = 0;
#my $temp_1 = ""; ## temporary to store $1
my $temp_2 = "";
our %mnemonics; ## hash to store mnemonics with opcodes
our %data_transfer;
our %branch;
our %manip;
my $line = 0;
my $sec_flag = "";
my $count = 0; ##holds total number of instructions
```

```
27 my $instruction_no;
28 my $instruction;
29 my $block_indicator = 0;
30 my $file_line = 0;
31 my $number ='';
32 our $sp_direct;
33 #gen
34 my $skipper;
35 my $no_inst = 1000; ##default number of instructions to be generated
36 my $write_handle;
37 my $outfile = "memory.t";
38 my $op1 = 0;
39 my $op2 = 0;
40 my $flag = 10;
41 my $oth_flag = 0;
42 my $data_flag = 0;
43 my $branch_flag = 0;
44 my $selection = "";
45 my @type = ("data_transfer", "manipulation", "branch");
46 my $temp_inst; my $temp_op1; my $temp_op2;
47 my $length; my $padding;
48 my @keys;
49 our $SP_top;
50 my $address_off;
51 my $IW1_flag = 0;
52 my $shize = 0;
53 my $repeat = 0;
54 my @ldstj;
55 my @jmp_loc;
56 my $flow;
57 my $condition;
58 my $jump;
59 my @keys_jmp;
60 our $dm_mask;
61 my @internal_stack;
62 my $routine_top;
63 my $call; ## keeps count of CALL in progress
64 my $routine_start; my $routine_end;
65 my $dynamic_no_inst;
66 my @criteria;
67 #mem_sizing
68 our $stack_per; ## percentage of overall memory
69 my $stack_size; ## Actual memory locations
70 our $mem_implemented;
71 our $custom_mem = 0; ##Flag to indicate custom Program memory (or both if VN
       )
72 our $custom_datamem = 0; ##Flag to indicate custom data memory
73 our $datamem_implemented;
74 my $data_memory;
```

```
75  my $mem_IO = 1; ## '1' = memory mapped IOs
76  my $reserved_space;
77  my $num_IO = 16; ##number of memory mapped IOs
78  my $fixed_data_space = 100; ##Fixed data memory space only in Von Neumann
79  my $usable_mem; ##Available program memory space
80  my $routine_space = 40; ##space reserved for sub-routines
81  my $fixed_jmp; ##helps to determine jump distance
82  #writeout
83  our $SP;
84  my $IW;
85  my $IW1;
86  my @reg; ## this mimics the registers
87  my @pm;  ## this mimics program memory
88  my @dm;  ## this mimics data memory -- internal
89  my @pm_out; ## Program memory to be written out
90  my @dm_out; ## Data memory to be written out
91  my @sf;  ## this mimics status flag
92  my @op_reg; ## All register data is store in this array. This is obsolete
        now in @reg_array
93  my @inst;
94  my @operand1;
95  my @operand2;
96  my @status_op; ##Array to store all status outputs
97  #calculation
98  my @bin;
99  my @temp_array;
100 my $calc_count;
101 my $temp_flag;
102 my $twos_test;
103 my $twos_comp;
104 my $twos_size;
105 my $neg_check;
106 my $operand1_reg;
107 my $operand2_reg;
108 #options
109 our $now = 0; ## '1'for PC relative, considering the current operation as
        refrence, '0' considering next operation as reference
110 my $debug_handle;
111 my $bug1; my $bug2;
112 my %jmp_cond = ("ju", '0000', "jc", '1000', "jneg", '0100', "jov", '0010', "
        jz", '0001', "jnc", '0111', "jpos", '1011', "jnov", '1101', "jnz", '1110'
        ); ## jump conditions
113 my $debug_handle1;
114 my $counter_call; ## my $inst_count = 0;
115 my $constant1;
116 my $inst_sequence;
117 my @reg_array; my @flag_array; my $sf_flag;
118 my $bck_jmp; my @bck_array;
119 my @srno_array; my $sr_no;
```

```
120
121 require "extract.pl";
122
123 if ($ARGV[0] =~ /-help/)
124 {
125         &help;
126 }
127 GetOptions ('config=s' => \$config, 'num=i' => \$number, 'mode=s' => \$mode)
        ;
128 print "Hello\nWorking....\n";
129 if ( $number ne "") { $no_inst = $number; print "NUMBER $number\n";}
130 if ( $mode eq "") { $mode = 'a'; }
131 else { $mode = $mode; }
132
133 if ($config eq "")
134 {
135         die "Configuration file not specified, going to exit\n";
136 }
137 else
138 {
139         $config = $user_input{-config};
140         print "$config \n";
141         open(my $read_handle, "<", "../$config") || die"died trying....
            couldn't open file:'$config'";
142         srand();
143         while(my $input = <$read_handle>)
144                 {
145                         ++${file_line};
146                         chomp($input);
147                         if($input =~ /^\s*#.*/) {next;} ## skipping comments
148                         elsif($input =~ /^\s*$/) {next;} ## skipping blank
                            lines
149                         elsif($input =~ /^\s*(\w*)\s*:\s*(\w*)\s*;*.*/)
150                         {
151                                 $flag = &extract($1,$2,$flag);
152                         }
153                         else { die "Invalid statement at $line\n";}
154                 }
155         print "NO of registers: $num_reg\n";
156         if(${count} > 0) { die "No. of instruction mapped > no. of
            instruction definitions by: $count \n"; }
157         $call = 0;
158         &memory_set; ##function to limit segment memory into allocated space
159         &filler; ##Fills program memory space with 0's and others with
            invalid data i.e. F's
160         $twos_test = '1'.(0 x (${bus_size}-1));
161         print "Twos Test: $twos_test\n";
162         $twos_test = oct("0b$twos_test");
```

```
163         $dm_mask = (($custom_datamem == 0)&&($arch == 1))? $twos_test :
                $dm_mask;
164         $count = 0; ##prep count to represent address in write back
165         $counter_call = 0;
166         &gen;
167         &write_out;
168         &reg_out;
169 }
170
171 ## 'memory_set' function checks and allocates memory space for program, data
        , srack and I/O
172 ## also warns if desired number of intrcutions exceed the number possible
173 sub memory_set
174 {
175         $memory_size = 2**$bus_size;
176         $constant1 = $memory_size;
177         if($custom_mem == 1) {$memory_size = $mem_implemented;} ## if memory
                implemeted is smaller than largest possible
178         print "memory size: $memory_size\n";
179         $data_memory = $memory_size;
180         if($custom_datamem == 1){$data_memory = $datamem_implemented} ##if
                data memory is smaller than largest possible
181         $temp_op1 = ($arch == 1)? ($stack_per * $memory_size) : ($stack_per
                * $data_memory);
182         $calc_count = $temp_op1 % 100;
183         $temp_op1 = $temp_op1 - $calc_count;
184         $stack_size = $temp_op1 / 100;
185         if($calc_count != 0){$stack_size = $stack_size + 1; }
186         if(($stack_size %2) != 0) {$stack_size = $stack_size + 1; }
187         print "Stack size: $stack_size\n";
188         $reserved_space = 0;
189         if(($mem_IO == 1) && ($arch == 1)) {
190                 $reserved_space = $num_IO;
191         }
192         if($arch == 1){
193                 $reserved_space = $reserved_space + $fixed_data_space +
                        $stack_size;
194         }
195         $reserved_space = $reserved_space + $routine_space + 2;
196         $usable_mem = $memory_size - $reserved_space;
197         $routine_top = ($arch == 1)? ($usable_mem + $fixed_data_space +2) :
                ($usable_mem + 2);
198         $routine_start = $routine_top;
199         if($no_inst > $usable_mem) {
200                 print "Warning: Number of instructions ($no_inst) greater
                        than available program memory size($usable_mem).\n";
201                 $no_inst = ($usable_mem-1);
202                 print "\t Only $no_inst number of instructions will be
                        created.\n";
```

```
203             }
204             $dynamic_no_inst = $no_inst;
205             print "Available program memory size($usable_mem).\n";
206     }
207     ##------- memory_set ends --
208
209     ##------- filler ----------
210     ## 'filler' function fills program memory space with 0's
211     ## All other memory space (including data memory) filled with 1's i.e. F's
212     sub filler
213     {
214             $sf[0] = 0; $sf[1] = 0;
215             $sf[2] = 0; $sf[3] = 0;
216             $sf_flag = 0;
217             $bck_jmp = 0;
218             $bck_array[0] = 0;
219             $bck_array[5] = 0;
220             $sr_no = 0;
221             $criteria[0] = 0;
222             $criteria[1] = 0;
223             $routine_end = $routine_start;
224             foreach my $i (0..$num_reg){
225                             $reg[$i] = 0;    }
226             $padding = (0 x ${bus_size});
227             $padding = sprintf("%X", oct("0b$padding"));
228             $calc_count = (1 x ${bus_size});
229             $calc_count = sprintf("%X", oct("0b$calc_count"));
230             foreach my $i(0..($memory_size-1)){
231                             $pm[$i] = 'k';
232                             $pm_out[$i] = 'k';
233                     $status_op[$i] = 'k';
234                     $jmp_loc[$i] = 'k';
235                     $inst[$i] = 'j';
236                     $operand1[$i] = 'j';
237                     $operand2[$i] = 'j';
238             }
239             if($arch == 0){
240                     for my $i (0..($data_memory-1)) {
241                             $dm[$i] = 'k';
242                             $dm_out[$i] = 'k'; }
243             }
244             foreach my $i (0..(($num_reg*$memory_size)-1)){
245                     $op_reg[$i] = 'k';
246                     $reg_array[$i] = 'k';
247             }
248             foreach my $i(0..$no_inst){
249                     $flag_array[$i] = 'k';
250                     $ldstj[$i] = 'k';
251             }
```

```
252 }
253 ##--- Filler Ends ----------
254
255 ##-------- write_out --------
256 ## writes the instructions and data memory to files
257 sub write_out{
258         print "WRITE_OUT $dynamic_no_inst\n";
259         $padding = (1 x $bus_size);
260         $padding = oct("0b$padding");
261         $padding = sprintf("%X", $padding);
262         foreach my $i (0..($memory_size-1)){
263                 $calc_count = sprintf("%X", ${i});
264                 if($i < $dynamic_no_inst){
265                 $bug1 = sprintf("%X", $pm_out[$i]);
266                 print $write_handle "\@${calc_count}\t$bug1 // $inst[$i]
                        $operand1[$i] $operand2[$i]\n";
267                 }
268                 elsif ($i <= $dynamic_no_inst){
269                         print $write_handle "\@${calc_count}\t$padding\n";
270                 }
271                 elsif ($i <= ($dynamic_no_inst +1)){
272                         print $write_handle "\@${calc_count}\t$padding\n";
273                 }
274                 else {
275                         if($pm_out[$i] eq 'k'){
276                                 print $write_handle "\@${calc_count}\
                                    t$padding\n";
277                         }
278                         else{
279                                 $bug1 = sprintf("%X", $pm_out[$i]);
280                                 print $write_handle "\@${calc_count}\t$bug1
                                    // $inst[$i] $operand1[$i] $operand2[$i]\
                                    n"; }
281                 }
282         }
283         if($arch == 0){
284                 open($write_handle, ">", "../data_memory.t") || die "died
                        trying... couldn't open file 'data_memory.t'";
285                 foreach my $i (0..($data_memory-1)){
286                         $calc_count = sprintf("%X", ${i});
287                         if($dm_out[$i] eq 'k'){
288                                 print $write_handle "\@${calc_count}\
                                    t$padding\n";
289                         }
290                         else{
291                                 $operand1_reg = sprintf("%X", $dm_out[$i]);
292                                 print $write_handle "\@${calc_count}\
                                    t$operand1_reg\n";
293                         }
```

```
294                 }
295         }
296         open($write_handle, ">", "reg.t") || die "died trying... couldn't
                open file 'reg.t'";
297         foreach my $i (0..(($num_reg*$no_inst)-1)){
298                 $calc_count = sprintf("%X", ${i});
299                 print $write_handle "\@${calc_count}\t$reg_array[$i]\n";
300         }
301 }
302 ##------- write_out Ends --
303
304 ##------- reg_out ---------
305 ## writes register values to files
306 sub reg_out
307 {
308         my $km;
309         my $ones;
310         my $output;
311         $ones = (1 x ($bus_size));
312         $ones = oct("0b$ones");
313         $ones = sprintf("%X", $ones);
314         foreach my $i (0 .. ($num_reg - 1)){
315                 $outfile = 'R'.$i.".t";
316                 $km = 0;
317                 $output = 0;
318                 open($write_handle, ">", $outfile) || die "died trying...
                        couldn't open file '$outfile'";
319                 foreach my $j (($i*$no_inst) .. (($i*$no_inst)+($no_inst-1))
                        ){
320                         $calc_count = sprintf("%X", ${km});
321                         if($km < ($no_inst - $bck_array[5])){
322                                 if(($km == 0) && ($reg_array[$j] eq 'k')){
323                                         $output = sprintf("%05X", 0); }
324                                 elsif($reg_array[$j] eq 'k'){
325                                         $output = $output; }
326                                 else { $output = $reg_array[$j];
327                                         $output = sprintf("%05X", $output
                                                ); }
328                                 print $write_handle "\@${calc_count}\
                                        t$output  // $srno_array[$km]\n";
329                         }
330                         ++$km;
331                 }
332         }
333         $outfile = "SR.t";
334         open($write_handle, ">", $outfile) || die "died trying... couldn't
                open file '$outfile'";
335         foreach my $i (0 .. ($no_inst - 1)){
336                 $calc_count = sprintf("%X", ${i});
```

```
337                 if(($i == 0)&&($flag_array[$i] eq 'k')){
338                         $output = 0;
339                         $temp_op1 = $srno_array[$i]; }
340                 elsif($flag_array[$i] eq 'k'){
341                                 $output = $output;
342                                 $temp_op1 = ""; }
343                 else { $output = $flag_array[$i];
344                                 $temp_op1 = $srno_array[$i]; }
345                 print $write_handle "\@${calc_count}\t$output   // $temp_op1
                        \n";
346         }
347         $outfile = "PC.t";
348         open($write_handle, ">", $outfile) || die "died trying... couldn't
                open file '$outfile'";
349         foreach my $i (0 .. ($no_inst - 1)){
350                 $calc_count = sprintf("%X", ${i});
351                 if($i < ($no_inst - $bck_array[5])){
352                         print $write_handle "\@${calc_count}\t$srno_array[$i
                                ]\n";
353                 }
354         }
355 }
356 ##-------- reg_out Ends --
357
358 ##-------- gen ------------
359 ## generates the required number of instructions
360 sub gen
361 {
362         $skipper = 0; $jump = 0; $flow = 0;
363         open($debug_handle, ">", "debug.txt") || die "died trying... couldn'
                t open file 'debug.txt'";
364         open($debug_handle1, ">", "debug1.txt") || die "died trying...
                couldn't open file 'debug1.txt'"; ###test
365         open($write_handle, ">", "../$outfile") || die "died trying...
                couldn't open file '$outfile'";
366         print "Instructions beign generated: $no_inst\n";
367         @keys = keys %mnemonics;
368         @keys_jmp = keys %jmp_cond;
369         my $inst_count = 0;
370         $instruction = 0;
371         my @temp_keys; ###jmp
372         @temp_keys = keys %manip; ###jmp
373         while($inst_count <= ($no_inst - 1)){
374                 ##print"\n";
375                 $inst_count = (($instruction eq "RET")&& (($mode =~ /b/)||(
                        $mode =~ /a/)) ) ? ++$inst_count: $inst_count;
376                 $selection = shuffle @keys;
377                 $instruction = $mnemonics{$selection};
378                 if($bck_jmp > 0){
```

```
379                         #++$counter;
380                         if($count == ($bck_array[2])){
381                                 $instruction = $mnemonics{JUMP};
382                                 ++$bck_jmp;
383                         }
384                         elsif($count == ($bck_array[3] - 2)){
385                                 $instruction = $mnemonics{JUMP};
386                                 ++$bck_jmp;
387                         }
388                         else{
389                                 while(($selection eq "JUMP")||($selection eq
                                        "RET")){##to block jumps inbetween jump-
                                    back pack
390                                     $selection = shuffle @keys;
391                                 }
392                                 $instruction = $mnemonics{$selection};
393                         }
394                 }
395                 if(($bck_jmp == 0)&&($jump == 1)&&(($jmp_loc[$count] eq 't')
                        ||($jmp_loc[$count] eq 't'))){ # to skip jump if jump
                        location has no instruction opcode
396                         $instruction = shuffle @temp_keys;
397                         $instruction_no = $manip{$instruction};
398                         $op1 = int(rand($num_reg-1));
399                         $op2 = int(rand($num_reg-1));
400                         &iw1_gen;
401                         ++$inst_count;
402                 }
403                 else{
404                         $jump = 0;
405                         if($manip{$instruction} && (($mode =~ /m/)||($mode
                            =~ /a/)) ){ #&& ($mode =~ )
406                             &manipulation_gen;
407                             while ($repeat == 1) {
408                                     @bin = keys %manip;
409                                     $instruction = shuffle @bin;
410                                     &manipulation_gen;
411                             }
412                         }
413                         elsif($data_transfer{$instruction} && (($mode =~ /d
                            /)||($mode =~ /a/)) ){
414                             if($inst_count <= ($no_inst - 2)){
415                                     &data_transfer;
416                             }
417                             else { $skipper = 1; }
418                             if($skipper != 1) { ++$inst_count; }
419                         }
420                         elsif($branch{$instruction} && (($mode =~ /b/)||(
                            $mode =~ /a/)) ){
```

```
421                                     if (($call != 0) && ($instruction eq "JUMP")
                                            ){
422                                             $skipper = 1;
423                                     }
424                                     elsif($instruction ne "RET"){
425                                             &branch;
426                                     }
427                                     else{
428                                             $instruction = "\0";
429                                             $skipper = 1;
430                                     }
431                                     if($skipper != 1) { ++$inst_count; }
432                             }
433                             else { $skipper = 1; }
434                             if($skipper == 1) { $skipper = 0;}
435                             else { ++$inst_count; }
436                     }
437             }
438     }
439     ##-------- gen Ends ----------
440
441     ##-------- branch ------------
442     ##function to generate branch instructions
443     sub branch{
444                     my $hex1; my $hex2; my $hex3;
445                     $instruction_no = $branch{$instruction};
446                     $op1 = int(rand($num_reg-1));
447                     if($bus_size == 12){
448                             $op1 = int(rand($op1_size-1)); ##in 12 bits Ri size
                                    = 2 in branch instructions
449                     }
450                     $op2 = int(rand($num_reg-1));
451                     $calc_count = sprintf("%X", $count);
452                     print $debug_handle "\@${calc_count}  $instruction \t";
453                     if($instruction eq "JUMP"){
454                             $IW1_flag = 1;
455                             while ($op1 == 2) { $op1 = int(rand($num_reg-1)); }
456                             $calc_count = 'jok';
457                             if(($op1 > 2) && ($reg[$op1] > ($memory_size - 1))){
458                                     $calc_count = 0;
459                                     for my $i (3..($num_reg-1)){
460                                             if($reg[$op1] < ($memory_size - 1)){
461                                                     $op1 = $i; $calc_count = '
                                                            jok';}
462                                     }
463                                     if($bus_size == 12){ $calc_count = 0; }
464                             }
465                             if($calc_count ne 'jok'){
466                                     $op1 = int(rand(1));
```

```
467                     }
468                     ##<-- Jump length calc -->##
469                     $fixed_jmp = ($dynamic_no_inst < 40) ? 3 : (
                            $dynamic_no_inst < 60) ? 4 :  ($dynamic_no_inst <
                             500) ? (5 + int(rand(6))): ($dynamic_no_inst <
                            1000) ? (5 + int(rand(11))) : (10 + int(rand(11))
                            );
470                     if($op1 == 0){
471                             $address_off = $count + $fixed_jmp + 2; ##
                                +2 because have to account for iw1 &Iw2
                                of jmp
472                             if($bck_jmp == 2){
473                                     print $debug_handle1 "count: $count
                                        ,, bck_jmp: $bck_jmp\n";
474                                     $address_off = $bck_array[4];
475                                     $fixed_jmp = $bck_array[4] - $count;
                                         ##reverse for negative jmp
                                        length
476                                     print $debug_handle1 "address_off:
                                        $address_off ,, fixed_jmp:
                                        $fixed_jmp ,,";
477                             }
478                             elsif($bck_jmp == 3){
479                                     print $debug_handle1 "count: $count
                                        ,, bck_jmp: $bck_jmp\n";
480                                     $address_off = $bck_array[3];
481                                     $fixed_jmp = $bck_array[3] - $count;
482                                     print $debug_handle1 "address_off:
                                        $address_off ,, fixed_jmp:
                                        $fixed_jmp ,,";
483                             }
484                             $bug1 = $address_off;
485                     }
486                     elsif ($op1 == 1){
487                             $address_off = $fixed_jmp;
488                             $bug1 = $count + $address_off;
489                             if($now == 0){ $bug1 = $bug1 + 2; }
490                             if($bck_jmp == 2){
491                                     $calc_count = ($now == 1) ? $count :
                                         ($count + 2);
492                                     $address_off = $bck_array[4];
493                                     $operand1_reg = $calc_count -
                                        $address_off;
494                                     $address_off = $memory_size -
                                        $operand1_reg;
495                                     $bug1 = $bck_array[4];
496                                     $fixed_jmp = $bck_array[4] -
                                        $calc_count; ##reverse for
                                        negative jmp length
```

```
497                                     print $debug_handle1 "count: $count
                                            ,, bck_jmp: $bck_jmp\n";
498                                     print $debug_handle1 "address_off:
                                            $address_off ,, fixed_jmp:
                                            $fixed_jmp ,,";
499                             }
500                             elsif($bck_jmp == 3){
501                                     $calc_count = ($now == 1) ? $count :
                                            ($count + 2);
502                                     $address_off = $bck_array[3] -
                                            $calc_count;
503                                     $fixed_jmp = $address_off;
504                                     $bug1 = $bck_array[3];
505                                     print $debug_handle1 "count: $count
                                            ,, bck_jmp: $bck_jmp\n";
506                                     print $debug_handle1 "address_off:
                                            $address_off ,, fixed_jmp:
                                            $fixed_jmp ,,";
507                             }
508                     }
509                     else{
510                             $address_off = $count + $fixed_jmp;
511                             if($bck_jmp == 2){
512                                     $address_off = $bck_array[4];
513                                     $fixed_jmp = $bck_array[4] - $count;
                                            ##reverse for negative jmp
                                            length
514                                     print $debug_handle1 "count: $count
                                            ,, bck_jmp: $bck_jmp\n";
515                                     print $debug_handle1 "address_off:
                                            $address_off ,, fixed_jmp:
                                            $fixed_jmp ,,";
516                             }
517                             elsif($bck_jmp == 3){
518                                     $address_off = $bck_array[3];
519                                     $fixed_jmp = $bck_array[3] - $count;
520                                     print $debug_handle1 "count: $count
                                            ,, bck_jmp: $bck_jmp\n";
521                                     print $debug_handle1 "address_off:
                                            $address_off ,, fixed_jmp:
                                            $fixed_jmp ,,";
522                             }
523                             $bug1 = $address_off;
524                             if($address_off < $reg[$op1] ){ ## address
                                    offset by calc roll-over
525                                     $operand1_reg = $reg[$op1] -
                                            $address_off;
526                                     $address_off = $memory_size -
                                            $operand1_reg;
```

```
527                                     }
528                                     else { $address_off = $address_off - $reg[
                                            $op1]; }
529                                     if($bck_jmp > 0){ print $debug_handle1 "
                                            after calc address_off: $address_off ,,";
                                             }
530                             }
531                             if($bug1 > ($dynamic_no_inst - 1)){
532                                     $skipper = 1;
533                                     $IW1_flag = 0; $flow = 0;
534                             }
535                             else{
536                                     print $debug_handle " operands: $op1 & $op2\
                                            n\tJmp length(d): $fixed_jmp ,, "; ##
                                            helps to determine jump distance
537                                     $hex1 = sprintf("%X",$bug1); $hex2 = sprintf
                                            ("%X",$address_off);
538                                     print $debug_handle "Final add: $hex1 ,,
                                            Address off: $hex2\t";
539                                     &jump_decide;
540                                     if($bck_jmp > 0){
541                                             while ($jump == 0){ &jump_decide; }
542                                     }
543                                     $flow = 1;
544
545                                     if(($no_inst > 60)&&($jump == 1)&&($call ==
                                            0)&&($bck_jmp == 0)&&($count >= ((
                                            $bck_array[0])*1000))){
546                                          ++$bck_jmp;
547                                          $criteria[0] = $criteria[0] + 1; ##
                                                Check if backward jump has
                                                occured
548                                          $bck_array[0] = $bck_array[0] + 1;
549                                          $bck_array[1] = $bug1;
550                                          $hex3 = sprintf("%X",$bck_array[1]);
551                                          print $debug_handle1 "bck_array[1]:
                                                $hex3 ,, ";
552                                          $bck_array[2] = $bug1 - 2;
553                                          $hex3 = sprintf("%X",$bck_array[2]);
554                                          print $debug_handle1 "count: $count
                                                ., bck_array[2]: $hex3 ,, ";
555                                          $fixed_jmp = 6 + int(rand(4));
556                                          $calc_count = $bug1 + $fixed_jmp;
557                                          $bck_array[3] = $calc_count;
558                                          $hex3 = sprintf("%X",$bck_array[3]);
559                                          print $debug_handle1 "bck_array[3]:
                                                $hex3 ,, ";
560                                          $calc_count = $count + 2;
561                                          $bck_array[4] = $calc_count;
```

```
562                                     $hex3 = sprintf("%X",$bck_array[4]);
563                                     print $debug_handle1 "bck_array[4]:
                                            $hex3\n\n";
564                             }
565                             else{
566                                     $jmp_loc[$bug1] = ($jump == 1)?
                                            $bug1 : 't'; ##to check if store
                                            jmp location with/out actual jump
567                             }
568                             &iw1_gen;
569                     }
570             } ## if for JMP
571             if($instruction eq "CALL"){
572                     while ($op1 == 2) { $op1 = int(rand($num_reg-1)); }
573                     $calc_count = 'cok';
574                     if(($op1 > 2) && ($reg[$op1] > ($memory_size - 1))){
575                             $calc_count = 0;
576                             for my $i (3..($num_reg-1)){
577                                     if($reg[$op1] < ($memory_size - 1)){
578                                             $op1 = $i; $calc_count = '
                                                    cok';}
579                             }
580                             if($bus_size == 12){ $calc_count = 0; }
581                     }
582                     if($calc_count ne 'cok'){
583                             $op1 = int(rand(1));
584                     }
585                     $fixed_jmp = ($call > 0)? (4 + int(rand(2))): (7 +
                            int(rand(3)));
586                     $address_off = $fixed_jmp + $routine_start;
587                     $calc_count = $routine_top + $routine_space -1; ##
                            calculating end of routine space
588                     if(($counter_call + 1 + $fixed_jmp) > ($no_inst - 2)
                            ){
589                             $skipper = 1; ##skipping when routine
                                    exceeds total instruction count
590                     }
591                     if($count > ($routine_end - 3)){
592                             $skipper = 1; ##skipping when call cannot be
                                    accommodated inside routine
593                     }
594                     if($address_off > ($routine_top + $routine_space -1)
                            ){
595                             $calc_count = $routine_top + $routine_space
                                    -1;
596                             $skipper = 1; #skipping when running out of
                                    routine space
597                     }
```

```
598                     if(($bck_jmp > 0)&&(($count > ($bck_array[3]-4))||(
                            $count > ($bck_array[1]-4)))){
599                         $skipper = 1; ##skipping CALL don't fit in
                                space between jumps
600                     }
601                     if(${sp_direct} == 1){  $calc_count = $SP_top -
                            $stack_size + 2; }
602                     else { $calc_count = $SP_top + $stack_size - 2; }
603                     if(($SP - $calc_count) < 0){ ### thsi will become (
                            $calc_count - $SP)< 0 if stack growing upwards
604                         $skipper = 1; #skipping when running out of
                                stack space
605                     }
606                     $skipper = ($call == 3) ? 1 : $skipper ; ##limiting
                            nested call to 3 levels
607                     if($skipper != 1){
608                         print $debug_handle "opearand: $op1\n";
609                         $hex1 = sprintf("%X", $routine_start);
610                         $counter_call = $counter_call + 2 +
                                $fixed_jmp;
611                         $dynamic_no_inst = $dynamic_no_inst -
                                $fixed_jmp; ## To have accurate end of
                                program ###14321
612                         $call = $call + 1;
613                         if($call > 1) {
614                             ++$counter_call; ##accounting for
                                    offset of nested call
615                             $criteria[1] = $criteria[1] + 1; ##
                                    Check if nested call has occured
616                         }
617                         if($op1 == 0){
618                             $address_off = $routine_start;
619                         }
620                         elsif ($op1 == 1){
621                             $calc_count = ($now == 1) ? $count :
                                    ($count + 2);
622                             if($calc_count >= $routine_start) {
                                    print $debug_handle1 "Error in
                                    CALL address this\n";}
623                             $address_off = $routine_start -
                                    $calc_count;
624                         }
625                         else{
626                             if($routine_start < $reg[$op1] ){
627                             $operand1_reg = $reg[$op1] -
                                    $routine_start;
628                             $address_off = $memory_size -
                                    $operand1_reg;
629                             }
```

```
630                                             else { $address_off = $routine_start
                                                    - $reg[$op1]; }
631                                         }
632                                         $hex3 = sprintf("%X", $address_off);
633                                         $calc_count = $count + 2;
634                                         $hex2 = sprintf("%X", $calc_count); ##PC in
                                                hex
635                                         push @internal_stack, $calc_count;
636                                         $calc_count = join('',@sf);
637                                         push @internal_stack, $calc_count; ##stored
                                                PC and Status flags
638                                         print $debug_handle "\t Stored, PC: $hex2 ,,
                                                 Status flag(ZVNC): @sf ,, ";
639                                         $calc_count = shuffle @keys_jmp;
640                                         $condition = $jmp_cond{$calc_count};
641                                         $flow = 1;
642                                         $IW1_flag = 1;
643                                         &iw1_gen;
644                                         $count = $routine_start; ##pointing to the
                                                start of routine
645                                         $routine_start = $fixed_jmp + $routine_start
                                                ;
646                                         $routine_end = $routine_start - 1;
647                                         push @internal_stack, $routine_end;
648                                         $calc_count = sprintf("%X", $routine_end);
649                                         $SP = ${sp_direct} ? (${SP} - 2):(${SP} + 2)
                                                ;
650                                         $hex2 = sprintf("%X", $SP);
651                                         print $debug_handle "SP: $hex2\n";
652                                         $hex2 = sprintf("%X", $routine_end);
653                                         print $debug_handle "\t Call to: $hex1 ,,
                                                routine length(d): $fixed_jmp ,, End of
                                                routine: $hex2 ,, Offset: $hex3";
654                                 }
655                         }##end if for CALL
656                         print $debug_handle "\n\n";
657 }
658 ##-------- branch Ends ----------
659
660 ##-------- jump_decide ----------
661 ## calculates wheather jump is taken depending on the condition
662 sub jump_decide
663 {
664                         $jump = 0;
665                         $calc_count = shuffle @keys_jmp;
666                         $instruction = $calc_count;
667                         $condition = $jmp_cond{$calc_count};
668                         print $debug_handle "Status flag(ZVNC) @sf ,, JMP type:
                                $calc_count";
```

```
669                 if($calc_count eq "ju"){
670                         $jump = 1;
671                 }
672                 elsif($calc_count eq "jc"){
673                         if ($sf[3] == 1) { $jump = 1; }
674                 }
675                 elsif($calc_count eq "jneg"){
676                         if ($sf[2] == 1) { $jump = 1; }
677                 }
678                 elsif($calc_count eq "jov"){
679                         if ($sf[1] == 1) { $jump = 1; }
680                 }
681                 elsif($calc_count eq "jz"){
682                         if ($sf[0] == 1) { $jump = 1; }
683                 }
684                 elsif($calc_count eq "jnc"){
685                         if ($sf[3] == 0) { $jump = 1; }
686                 }
687                 elsif($calc_count eq "jpos"){
688                         if ($sf[2] == 0) { $jump = 1; }
689                 }
690                 elsif($calc_count eq "jnov"){
691                         if ($sf[1] == 0) { $jump = 1; }
692                 }
693                 elsif($calc_count eq "jnz"){
694                         if ($sf[0] == 0) { $jump = 1; }
695                 }
696                 else{
697                         $jump = 0;
698                         print "ERROR: Wrong jump conditon detected\n";
699                 }
700                 $calc_count = ($jump == 1) ? "Taken" : "Not taken" ;
701                 print $debug_handle " $calc_count";
702 }
703 ##------- jump_decide Ends --------
704
705 ##------- data_transfer ----------
706 #function to generate data_transfer instructions
707 sub data_transfer{
708                  my $hex1; my $hex2; my $hex3;
709                 $instruction_no = $data_transfer{$instruction};
710                 $calc_count = sprintf("%X", $count);
711                 print $debug_handle "\@${calc_count}\t$instruction  ";
712                 if($call !=0 ){
713                         while ($op1 == 2) { $op1 = int(rand($num_reg - 1));
                            } ## comment out this line to generate all
                            addressing modes, mode 2 blocked
714                 }
715                 $op2 = int(rand($num_reg-1));
```

```
716                 if(($call > 0)&&($count > ($routine_end - 2))){
717                         $skipper = 1; ##skipping when LOAD & STORE cannot be
                                accommodated inside routine
718                 }
719                 if(($bck_jmp > 0)&&(($count > ($bck_array[3]-4))||($count >
                        ($bck_array[1]-4)))){
720                         $skipper = 1; ##skipping when LOAD & STORE don't fit
                                in space between jumps
721                 }
722                 if($skipper != 1){
723                         if($instruction eq "LOAD"){
724                                 $IW1_flag = 1;
725                                 if($op1 > 2){
726                                         $op1 = 0;
727                                    while ($op1 < 3){ $op1 = int(rand(${
                                           num_reg}-1)); } #to generate reg >= 3
728                                      if ($arch == 1){
729                                             if($reg[$op1] > ($usable_mem
                                                 + $fixed_data_space - 1)
                                                ){
730                                                  for my $i (3..(
                                                      $num_reg-1)){
731                                                       if($reg[$op1
                                                           ] < (
                                                           $usable_mem
                                                            +
                                                           $fixed_data_space
                                                            - 1))
732                                                       { $op1 = $i;

                                                           $calc_count
                                                            = 'ok';}
733                                                  }
734                                             }
735                                             else{
736                                                  $calc_count = 'ok';
                                                     }
737                                      }
738                                      else{
739                                             if($reg[$op1] > (
                                                 $data_memory - 1)){
740                                                  for my $i (3..(
                                                      $num_reg-1)){
741                                                       if($reg[$op1
                                                           ] < (
                                                           $data_memory
                                                            - 1))
742                                                       { $op1 = $i;
```

```
                                                                $calc_count
                                                                 = 'ok';}
743                                                                     }
744                                                             }
745                                                             else{
746                                                                     $calc_count = 'ok';
                                                                        }
747                                                     }
748                                                     if($calc_count ne 'ok'){
749                                                             $op1 = int(rand(1)); ##
                                                                Restricted to direct or
                                                                pc-relative
750                                                     }
751                                             }
752                                             if($sp_direct == 1){    $calc_count =
                                                $SP_top - $stack_size; }
753                                             else { $calc_count = $SP_top + $stack_size;
                                                }
754                                             if(($SP == $SP_top)&&($op1 == 2)) { $op1 =
                                                int(rand(1)); }
755                                             print $debug_handle "opearnds: $op1 & $op2\t
                                                ";
756                                             if($op1 == 0){ ## direct addressing
757                                             print $debug_handle "Direct addressing\t";
758                                                     if($arch == 1){
759                                                             $address_off = 0;
760                                                             while ($address_off <= (
                                                                $no_inst + 1)){
                                                                $address_off = int(rand($
                                                                {usable_mem}+${
                                                                fixed_data_space}));} ##
                                                                load address restricted
                                                                to address after
                                                                instructions
761                                                             if($pm[$address_off] eq 'k')
                                                                {
762                                                                     $pm[$address_off] =
                                                                        int(rand((2**${
                                                                        bus_size})-1));
763                                                                     $pm_out[$address_off
                                                                        ] = $pm[
                                                                        $address_off];
764                                                             }
765                                                             $reg[$op2] = $pm[
                                                                $address_off]; ##load
                                                                operation
766                                                     }
767                                                     else {
```

```
768                                     $address_off = int(rand(
                                            $data_memory -1 -
                                            $stack_size - $num_IO));
769                                     if($dm[$address_off] eq 'k')
                                            {
770                                             $dm[$address_off] =
                                                int(rand((2**${
                                                bus_size})-1));
771                                             $dm_out[$address_off
                                                ] = $dm[
                                                $address_off];
772                                     }
773                                     $reg[$op2] = $dm[
                                            $address_off];
774                                  }
775                                 $hex1 = sprintf("%X", $address_off);
                                        $hex2 = sprintf("%X", $reg[$op2
                                    ]);
776                                 print $debug_handle "Address: $hex1
                                    ,, Value: $hex2";
777                         }
778                         elsif($op1 == 1) { ## PC-relative
779                         print $debug_handle "PC-relative\t";
780                                 if($arch == 1){
781                                     $address_off = 0;
782                                     while ($address_off <= (
                                            $no_inst + 1)){
                                            $address_off = int(rand($
                                            {usable_mem}+${
                                            fixed_data_space}));}
783                                     if($pm[$address_off] eq 'k')
                                            {
784                                             $pm[$address_off] =
                                                int(rand((2**${
                                                bus_size})-1));
785                                             $pm_out[$address_off
                                                ] = $pm[
                                                $address_off];
786                                     }
787                                     $reg[$op2] = $pm[
                                            $address_off]; ##load
                                            operation
788                                 }
789                                 else {
790                                     $address_off = int(rand(
                                            $data_memory -1 -
                                            $stack_size - $num_IO));
791                                     if($dm[$address_off] eq 'k')
                                            {
```

```
792                                                 $dm[$address_off] =
                                                        int(rand((2**${
                                                        bus_size})-1));
793                                                 $dm_out[$address_off
                                                        ] = $dm[
                                                        $address_off];
794                                             }
795                                             $reg[$op2] = $dm[
                                                    $address_off];
796                                         }
797                                         $hex1 = sprintf("%X", $address_off);
                                                $hex2 = sprintf("%X", $reg[$op2
                                                ]);
798                                         print $debug_handle "Address(orig):
                                                $hex1 ,, Value: $hex2\t";
799                                         if($now == 1) { $calc_count = $count
                                                ;}
800                                         else {$calc_count = (${count} + 2);}
801                                         if($address_off < $calc_count ){ ##
                                                address offset by calc roll-over
802                                             $operand1_reg = $calc_count
                                                    - $address_off;
803                                             $address_off = $constant1 -
                                                    $operand1_reg;
804                                         }
805                                         else { $address_off = $address_off -
                                                $calc_count; }
806                                         $hex1 = sprintf("%X", $calc_count);
                                                $hex2 = sprintf("%X",
                                                $address_off);
807                                         print $debug_handle "PC: $hex1\
                                                tOffset: $hex2";
808                                     }
809                                     elsif($op1 == 2) { ## SP
810                                     print $debug_handle "Stack Addressing\t";
811                                             $address_off = 0;
812                                             $SP = ($SP == $SP_top)? $SP : (${
                                                sp_direct}? (${SP} + 1):(${SP} -
                                                1));
813                                             $calc_count = ($arch == 0)? ($SP &
                                                $dm_mask) : $SP;
814                                             if($arch == 1){
815                                                 if($pm[$calc_count] eq 'k'){
816                                                     $pm[$calc_count] =
                                                        0;
817                                                     $pm_out[$calc_count]
                                                        = $pm[
                                                        $calc_count];
818                                                 }
```

```
819                                     $reg[$op2] = $pm[$calc_count
                                            ]; ##load operation
820                                     $hex1 = sprintf("%X", $SP);
                                            $hex2 = sprintf("%X",
                                            $calc_count); $hex3 =
                                            sprintf("%X", $reg[$op2])
                                            ;
821                                     print $debug_handle "
                                            St_pointer: $hex1 ,, pm[
                                            $hex2]: $hex3";
822                                 }
823                                 else{
824                                     if($dm[$calc_count] eq 'k'){
825                                         $dm[$calc_count] =
                                                0;
826                                         $dm_out[$calc_count]
                                                = $dm[
                                                $calc_count];
827                                     }
828                                     $reg[$op2] = $dm[$calc_count
                                            ]; ##load operation
829                                     $hex1 = sprintf("%X", $SP);
                                            $hex2 = sprintf("%X",
                                            $calc_count); $hex3 =
                                            sprintf("%X", $reg[$op2])
                                            ;
830                                     print $debug_handle "
                                            St_pointer: $hex1 ,, dm[
                                            $hex2]: $hex3";
831                                 }
832                             }
833                             else{
834                               print $debug_handle "Register Addressing\t
                                      ";
835                                 if($arch == 1){
836                                     if($reg[$op1] <= ($no_inst
                                            +1)){
837                                         $address_off =
                                                $no_inst + 3;
838                                         while($address_off
                                                <= (($no_inst +
                                                1)-$reg[$op1])) {
                                                 $address_off =
                                                int(rand(($${
                                                usable_mem}+$${
                                                fixed_data_space
                                                })-$reg[$op1]));}
839                                     }
840                                     else{
```

```
841                                         $address_off = int(
                                                rand(($${
                                                usable_mem}+$${
                                                fixed_data_space
                                                })-$reg[$op1]));
842                                     }
843                                     $calc_count = $reg[$op1] +
                                            $address_off;
844                                     if($pm[$calc_count] eq 'k'){
845                                         $pm[$calc_count] =
                                                int(rand((2**$${
                                                bus_size})-1));
846                                         $pm_out[$calc_count]
                                                 = $pm[
                                                $calc_count];
847                                     }
848                                     $reg[$op2] = $pm[$calc_count
                                            ];
849                                     $hex1 = sprintf("%X",
                                            $calc_count); $hex2 =
                                            sprintf("%X",
                                            $address_off); $hex3 =
                                            sprintf("%X", $reg[$op2])
                                            ;
850                                     print $debug_handle "Address
                                            : $hex1 ,, Offset: $hex2
                                            ,, Value: $hex3";
851                                 }
852                                 else {
853                                     $address_off = int(rand(
                                            $data_memory - 1 -
                                            $stack_size - $num_IO));
854                                     if($dm[$address_off] eq 'k')
                                            {
855                                         $dm[$address_off] =
                                                int(rand((2**$${
                                                bus_size})-1));
856                                         $dm_out[$address_off
                                                ] = $dm[
                                                $address_off];
857                                     }
858                                     $operand2_reg = $address_off
                                            ;
859                                     if($address_off < $reg[$op1
                                            ]){
860                                         $operand1_reg = $reg
                                                [$op1] -
                                                $address_off;
```

```
861                                                     $address_off =
                                                            $data_memory -
                                                            $operand1_reg;
862                                                 }
863                                                 else { $address_off =
                                                        $address_off - $reg[$op1
                                                        ]; }
864                                                 $reg[$op2] = $dm[
                                                        $operand2_reg];
865                                                 $hex1 = sprintf("%X",
                                                        $operand2_reg); $hex2 =
                                                        sprintf("%X",
                                                        $address_off); $hex3 =
                                                        sprintf("%X", $reg[$op2])
                                                        ;
866                                                 print $debug_handle "Address
                                                        : $hex1 ,, Offset: $hex2
                                                        ,, Value: $hex3";
867                                         }
868                                 }
869                         print $debug_handle "\n\n";
870                         $ldstj[$count] = $count; ## Storing load location to
                                be used by jump
871                         $op_reg[($op2*$memory_size)+($count+1)] = $reg[$op2
                            ];### <--- storing the value in reg file
872                         $operand1_reg = ($op2*$no_inst)+($count+1);
873                         }
874                         elsif($instruction eq "STORE"){
875                                 $IW1_flag = 1;
876                                 if($op1 > 2){
877                                         $op1 = 0;
878                                    while ($op1 < 3){ $op1 = int(rand(${
                                        num_reg}-1)); }
879                                         if ($arch == 1){
880                                                 if($reg[$op1] > ($usable_mem
                                                     + $fixed_data_space - 1)
                                                     ){
881                                                         for my $i (3..(
                                                             $num_reg-1)){
882                                                                 if($reg[$op1
                                                                     ] < (
                                                                     $usable_mem
                                                                      +
                                                                     $fixed_data_space
                                                                      - 1))
883                                                                 { $op1 = $i;

                                                                     $calc_count
                                                                      = 'ok';}
```

```
884                                                     }
885                                             }
886                                             else{
887                                                     $calc_count = 'ok';
                                                        }
888                                     }
889                                     else{
890                                             if($reg[$op1] > (
                                                $data_memory - 1)){
891                                                     for my $i (3..(
                                                        $num_reg-1)){
892                                                             if($reg[$op1
                                                                ] < (
                                                                $data_memory
                                                                 - 1))
893                                                             { $op1 = $i;

                                                                $calc_count
                                                                 = 'ok';}
894                                                     }
895                                             }
896                                             else{
897                                                     $calc_count = 'ok';
                                                        }
898                                     }
899                                     if($calc_count ne 'ok'){
900                                             $op1 = int(rand(1)); ##
                                                Restricted to direct or
                                                pc-relative
901                                     }
902                             }
903                             if(${sp_direct} == 1){  $calc_count =
                                $SP_top - $stack_size + 1; }
904                             else { $calc_count = $SP_top + $stack_size -
                                 1; }
905                             if((($SP - $calc_count) < 0)&&($op1 == 2)) {
                                 $op1 = int(rand(1)); }
906                             if($op1 == 0){ ## direct addressing
907                             print $debug_handle "Direct addressing\t";
908                                     if($arch == 1){
909                                             $address_off = 0;
910                                             while ($address_off <= (${
                                                no_inst}+1)){
                                                $address_off = int(rand($
                                                {usable_mem}+${
                                                fixed_data_space}));} ##
                                                load address restricted
                                                to address after
                                                instructions
```

```
911                                     $pm[$address_off] = $reg[
                                            $op2]; ##store operation
912                                     $hex1 = sprintf("%X",
                                            $address_off); $hex2 =
                                            sprintf("%X", $pm[
                                            $address_off]);
913                                     print $debug_handle "Address
                                            : $hex1 ,, store Value:
                                            $hex2 in \$pm[$hex1] ";
914                             }
915                             else {
916                                     $address_off = int(rand(
                                            $data_memory -1 -
                                            $stack_size - $num_IO));
917                                     $dm[$address_off] = $reg[
                                            $op2];
918                                     $hex1 = sprintf("%X",
                                            $address_off); $hex2 =
                                            sprintf("%X", $dm[
                                            $address_off]);
919                                     print $debug_handle "Address
                                            : $hex1 ,, store Value:
                                            $hex2 in \$dm[$hex1] ";
920                             }
921                     }
922                     elsif($op1 == 1) { ## PC-relative
923                     ##print "PC-relative\n";
924                     print $debug_handle "PC-relative\t";
925                             if($arch == 1){
926                                     $address_off = 0;
927                                     while ($address_off <= (
                                            $no_inst + 1)){
                                            $address_off = int(rand($
                                            {usable_mem}+${
                                            fixed_data_space}));}
928                                     $pm[$address_off] = $reg[
                                            $op2];
929                                     $hex1 = sprintf("%X",
                                            $address_off); $hex2 =
                                            sprintf("%X", $pm[
                                            $address_off]);
930                                     print $debug_handle "Address
                                            (orig): $hex1 ,, store
                                            Value: $hex2 in \$pm[
                                            $hex1] ";
931                             }
932                             else {
933                                     $address_off = int(rand(
                                            $data_memory -1 -
```

```
                                        $stack_size - $num_IO));
934                                     $dm[$address_off] = $reg[
                                            $op2];
935                                     $hex1 = sprintf("%X",
                                            $address_off); $hex2 =
                                            sprintf("%X", $reg[$op2])
                                            ;
936                                     print $debug_handle "Address
                                            (orig): $hex1 ,, store
                                            Value: $hex2 in \$dm[
                                            $hex1] ";
937                                 }
938                                 if($now == 1) { $calc_count = $count
                                        ;}
939                                 else {$calc_count = (${count} + 2);}
940                                 if($address_off < $calc_count ){ ##
                                        making it roll over - calc
941                                     $operand1_reg = $calc_count
                                            - $address_off;
942                                     $address_off = $constant1 -
                                            $operand1_reg;
943                                 }
944                                 else { $address_off = $address_off -
                                         $calc_count; }
945                                 $hex1 = sprintf("%X", $calc_count);
                                        $hex2 = sprintf("%X",
                                        $address_off);
946                                 print $debug_handle "PC: $hex1\
                                        tOffset: $hex2";
947                             }
948                             elsif($op1 == 2) { ## SP
949                                 print $debug_handle "Stack
                                        Addressing\t";
950                                 $address_off = 0;
951                                 print "$SP\n";
952                                 $calc_count = ($arch == 0)? ($SP &
                                        $dm_mask) : $SP;
953                                 if($arch == 1) {
954                                     $pm[$calc_count] = $reg[$op2
                                            ]; ##store operation
955                                     $hex1 = sprintf("%X", $SP);
                                            $hex2 = sprintf("%X",
                                            $calc_count); $hex3 =
                                            sprintf("%X", $reg[$op2])
                                            ;
956                                     print $debug_handle "
                                            St_pointer: $hex1 ,, pm[
                                            $hex2]: $hex3";
957                                 }
```

```
958                                     else{
959                                             $dm[$calc_count] = $reg[$op2
                                                ]; ##store operation
960                                             $hex1 = sprintf("%X", $SP);
                                                $hex2 = sprintf("%X",
                                                $calc_count); $hex3 =
                                                sprintf("%X", $reg[$op2])
                                                ;
961                                             print $debug_handle "
                                                St_pointer: $hex1 ,, dm[
                                                $hex2]: $hex3";
962                                     }
963                                     $SP = ${sp_direct} ? (${SP} - 1):(${
                                            SP} + 1);
964                             }
965                             else{
966                                     print $debug_handle "Register
                                            Addressing\t";
967                                     if($arch == 1){
968                                             if($reg[$op1] <= ($no_inst +
                                                    1)){
969                                                     $address_off = ${
                                                        no_inst}+3;
970                                                     while($address_off
                                                        <= (($no_inst +
                                                        1)-$reg[$op1])) {
                                                        $address_off =
                                                        int(rand((${
                                                        usable_mem}+${
                                                        fixed_data_space
                                                        })-$reg[$op1]));}
971                                             }
972                                             else{
973                                                     $address_off = int(
                                                        rand((${
                                                        usable_mem}+${
                                                        fixed_data_space
                                                        })-$reg[$op1]));
974                                             }
975                                             $calc_count = $reg[$op1] +
                                                    $address_off;
976                                             $pm[$calc_count] = $reg[$op2
                                                ];
977                                             $hex1 = sprintf("%X",
                                                $calc_count); $hex2 =
                                                sprintf("%X",
                                                $address_off); $hex3 =
                                                sprintf("%X", $reg[$op2])
                                                ;
```

```
978                                                 print $debug_handle "Address
                                                        : $hex1 ,, Offset: $hex2
                                                        ,, Value: $hex3";
979                                             }
980                                             else {
981                                                 $address_off = int(rand(
                                                        $data_memory - 1 -
                                                        $stack_size - $num_IO));
982                                                 $dm[$address_off] = $reg[
                                                        $op2];
983                                                 $hex1 = sprintf("%X",
                                                        $address_off);
984                                                 ###
985                                                 if($address_off < $reg[$op1
                                                        ]){
986                                                     $operand1_reg = $reg
                                                            [$op1] -
                                                            $address_off;
987                                                     $address_off =
                                                            $data_memory -
                                                            $operand1_reg;
988                                                 }
989                                                 else { $address_off =
                                                        $address_off - $reg[$op1
                                                        ]; }
990                                                 ###
991                                                 $hex2 = sprintf("%X",
                                                        $address_off); $hex3 =
                                                        sprintf("%X", $reg[$op2])
                                                        ;
992                                                 print $debug_handle "Address
                                                        : $hex1 ,, Offset: $hex2
                                                        ,, Value: $hex3";
993                                             }
994                                         }
995                                 print $debug_handle "\n\n";
996                                 $ldstj[$count] = $count;
997                                 } ##-- Store Ends ---
998                                 &iw1_gen;
999             }#if for skipper condition
1000    }
1001    ##-------- data_transfer Ends ---------
1002
1003    ##-------- manipulation_gen ----------
1004    #function to generate maipulation instructions
1005    sub manipulation_gen {
1006                    $repeat = 0;
1007                    $instruction_no = $manip{$instruction};
1008                    $calc_count = sprintf("%X", $count);
```

```
1009                print $debug_handle "\@${calc_count}\t$instruction\t";
1010                $op1 = int(rand($num_reg-1));
1011                if(($instruction eq "ADD") or ($instruction eq "SUB") or (
                        $instruction eq "MUL") or ($instruction eq "DIV")  or (
                        $instruction eq "NOT") or
1012                       ($instruction eq "AND") or ($instruction eq "OR")  or (
                           $instruction eq "XOR") or ($instruction eq "COPY") or
                           ($instruction eq "SWAP")){
1013                          $op2 = int(rand($num_reg-1)); #correct register
                              value can also be obtained by using 2^
                              operand2_size - 1 if register space is different
                              from op1
1014                }
1015                else{
1016                        $op2 = int(rand((2**${op2_size})-1));
1017                }
1018                if($instruction eq "ADD"){
1019                        $bug1 = sprintf("%X", $reg[$op1]);
1020                        $bug2 = sprintf("%X", $reg[$op2]);
1021                        print $debug_handle "$op1 \[$bug1\]\t$op2 \[$bug2\]\
                            t";
1022                        $operand1_reg = $reg[$op1]; $operand2_reg = $reg[
                            $op2];
1023                        $reg[$op1] = $reg[$op1] + $reg[$op2];
1024                        &CNVZ;
1025                        &compactor;
1026                        $bug1 = sprintf("%X", $reg[$op1]);
1027                        $op_reg[($op1*$memory_size)+$count] = $reg[$op1];
1028                }
1029                elsif($instruction eq "ADDC"){
1030                        $bug1 = sprintf("%X", $reg[$op1]);
1031                        $bug2 = sprintf("%X", $op2);
1032                        print $debug_handle "$op1 \[$bug1\]\t$op2 \[$bug2\]\
                            t";
1033                        $operand1_reg = $reg[$op1]; $operand2_reg = $op2;
1034                        $reg[$op1] = $reg[$op1] + $op2;
1035                        &CNVZ;
1036                        &compactor;
1037                        $bug1 = sprintf("%X", $reg[$op1]);
1038                        print $debug_handle "Res: $bug1\n\n";
1039                        $op_reg[($op1*$memory_size)+$count] = $reg[$op1];
1040                }
1041                elsif($instruction eq "SUB"){
1042                        $bug1 = sprintf("%X", $reg[$op1]);
1043                        $bug2 = sprintf("%X", $reg[$op2]);
1044                        print $debug_handle "$op1 \[$bug1\]\t$op2 \[$bug2\]\
                            t";
1045                        $operand1_reg = $reg[$op1]; $operand2_reg = $reg[
                            $op2];
```

```
1046                    $reg[$op1] = $reg[$op1] - $reg[$op2];
1047                    &CNVZ;
1048                    &compactor;
1049                    $bug1 = sprintf("%X", $reg[$op1]);
1050                    print $debug_handle "Res: $bug1\n\n";
1051                    $op_reg[($op1*$memory_size)+$count] = $reg[$op1];
1052            }
1053            elsif($instruction eq "SUBC"){
1054                    $bug1 = sprintf("%X", $reg[$op1]);
1055                    $bug2 = sprintf("%X", $op2);
1056                    print $debug_handle "$op1 \[$bug1\]\t$op2 \[$bug2\]\
                           t";
1057                    $operand1_reg = $reg[$op1]; $operand2_reg = $op2;
1058                    $reg[$op1] = $reg[$op1] - $op2;
1059                    &CNVZ;
1060                    &compactor;
1061                    $bug1 = sprintf("%X", $reg[$op1]);
1062                    print $debug_handle "Res: $bug1\n\n";
1063                    $op_reg[($op1*$memory_size)+$count] = $reg[$op1];
1064            }
1065            elsif($instruction eq "MUL"){
1066                    $bug1 = sprintf("%X", $reg[$op1]);
1067                    $bug2 = sprintf("%X", $reg[$op2]);
1068                    print $debug_handle "$op1 \[$bug1\]\t$op2 \[$bug2\]\
                           t";
1069                    $neg_check = 0;
1070                    if($reg[$op1]>= $twos_test){
1071                            ++$neg_check;
1072                            $twos_comp = $reg[$op1];
1073                            $twos_size = $bus_size;
1074                            &complement;
1075                            $reg[$op1] = oct("0b$twos_comp");
1076                    }
1077                    if($reg[$op2]>= $twos_test){
1078                            ++$neg_check;
1079                            $twos_comp = $reg[$op2];
1080                            $twos_size = $bus_size;
1081                            &complement;
1082                            $reg[$op2] = oct("0b$twos_comp");
1083                    }
1084                    $calc_count = $reg[$op1] * $reg[$op2];
1085                    if($calc_count == 0){ $sf[0] = 1; } ## Zero Flag
1086                    else { $sf[0] = 0; }
1087                    if(($neg_check == 1)&&($op1 != $op2)){
1088                            $twos_comp = $calc_count;
1089                            $twos_size = 2*$bus_size;
1090                            &complement;
1091                            $calc_count = $twos_comp;
1092                    }
```

```
1093                    else{
1094                            $calc_count = sprintf("%b", $calc_count);
1095                            $length = length($calc_count);
1096                            if($length > (2*$bus_size)){
1097                                    print "Warning: Length exceeding ${
                                           twos_size} at instruction number
                                           $count : $instruction\n";
1098                            }
1099                            $padding = (0 x ((2*${bus_size}) - $length))
                                   ;
1100                            $calc_count = $padding.$calc_count;
1101                    }
1102                    $neg_check = 0;
1103                    @temp_array = split('',$calc_count);
1104                    @temp_array = reverse(@temp_array);
1105                    @temp_array = @temp_array[0 .. ((2*${bus_size})-1)];
1106                    @temp_array = reverse(@temp_array);
1107                    @bin = @temp_array[0 .. (${bus_size} -1)]; #TALUH
1108                    $calc_count = join('', @bin);
1109                    $reg[$op1] = oct("0b$calc_count");
1110                    $op_reg[($op1*$memory_size)+$count] = $reg[$op1];
1111                    $bug1 = sprintf("%X", $reg[$op1]);
1112                    print $debug_handle "Res1:$op1 \[$bug1\]\t";
1113                    @bin = @temp_array[${bus_size} .. ((2*${bus_size})
                           -1)]; #TALUL
1114                    $calc_count = join('', @bin);
1115                    $reg[$op2] = oct("0b$calc_count");
1116                    $op_reg[($op2*$memory_size)+$count] = $reg[$op2];
1117                    $bug2 = sprintf("%X", $reg[$op2]);
1118                    print $debug_handle "Res2:$op2 \[$bug2\]\n\n";
1119            }
1120            elsif($instruction eq "MULC"){
1121                    $bug1 = sprintf("%X", $reg[$op1]);
1122                    $bug2 = sprintf("%X", $op2);
1123                    print $debug_handle "$op1 \[$bug1\]\t$op2 \[$bug2\]\
                           t";
1124                    $neg_check = 0;
1125                    if($reg[$op1]>= $twos_test){
1126                            ++$neg_check;
1127                            $twos_comp = $reg[$op1];
1128                            $twos_size = $bus_size;
1129                            &complement;
1130                            $reg[$op1] = oct("0b$twos_comp");
1131                    }
1132                    $calc_count = $reg[$op1] * $op2;
1133                    if($calc_count == 0){ $sf[0] = 1; } ## Zero Flag
1134                    else { $sf[0] = 0; }
1135                    if($neg_check == 1){
1136                            $twos_comp = $calc_count;
```

```
1137                                    $twos_size = 2*$bus_size;
1138                                    &complement;
1139                                    $calc_count = $twos_comp;
1140                            }
1141                            else{
1142                                    $calc_count = sprintf("%b", $calc_count);
1143                                    $length = length($calc_count);
1144                                    if($length > (2*$bus_size)){
1145                                            print "Warning: Length exceeding ${
                                                twos_size} at instruction number
                                                $count : $instruction\n"; }
1146                                    $padding = (0 x ((2*${bus_size}) - $length))
                                            ;
1147                                    $calc_count = $padding.$calc_count;
1148                            }
1149                            $neg_check = 0;
1150                            @temp_array = split('',$calc_count);
1151                            @temp_array = reverse(@temp_array);
1152                            @temp_array = @temp_array[0 .. ((2*${bus_size})-1)];
1153                            @temp_array = reverse(@temp_array);
1154                            $calc_count = join('',@temp_array);
1155                            @bin = @temp_array[0 .. (${bus_size} -1)];
1156                            $calc_count = join('', @bin);
1157                            $reg[$op1] = oct("0b$calc_count"); #TALUH
1158                            $op_reg[($op1*$memory_size)+$count] = $reg[$op1];
1159                            $bug1 = sprintf("%X", $reg[$op1]);
1160                            print $debug_handle "Res1:$op1 \[$bug1\]\n\n";
1161                            }
1162                    elsif($instruction eq "DIV"){
1163                            if(($reg[$op2] == 0) || ($reg[$op2] == $twos_test)){
                                    ##Div by 0 avoided
1164                                    $shize = 1; $repeat = 1;
1165                            }
1166                            else{
1167                                    $bug1 = sprintf("%X", $reg[$op1]);
1168                                    $bug2 = sprintf("%X", $reg[$op2]);
1169                                    print $debug_handle "$op1 \[$bug1\]\t$op2 \[
                                            $bug2\]\t";
1170                                    $neg_check = 0;
1171                                    if($reg[$op1]>= $twos_test){
1172                                            ++$neg_check;
1173                                            $twos_comp = $reg[$op1];
1174                                            $twos_size = $bus_size;
1175                                            &complement;
1176                                            $reg[$op1] = oct("0b$twos_comp");
1177                                            $operand1_reg = $reg[$op1];
1178                                            $bug1 = "neg";
1179                                    }
1180                                    if($reg[$op2]>= $twos_test){
```

```
1181                                    ++$neg_check;
1182                                    $twos_comp = $reg[$op2];
1183                                    $twos_size = $bus_size;
1184                                    &complement;
1185                                    $reg[$op2] = oct("0b$twos_comp");
1186                                    $bug2 = "neg";
1187                            }
1188                            $operand2_reg = $reg[$op2];
1189                            if(($reg[$op2] > $reg[$op1])&&($reg[$op1] !=
                                    0)){
1190                                    $calc_count = (1 x $bus_size);
1191                                    $calc_count = oct("0b$calc_count");
1192                                    $reg[$op2] = ($bug1 eq "neg") ? (
                                        $reg[$op2] - $reg[$op1]) : $reg[
                                        $op1] ;
1193                                    $reg[$op1] = ($bug1 ne "neg") ? 0 :
                                        (($bug2 eq "neg") ? 1 :
                                        $calc_count);
1194                                    $bug1 = 0; $bug2 = 0;
1195                                    $op_reg[($op2*$memory_size)+$count]
                                        = $reg[$op2];
1196                                    $op_reg[($op1*$memory_size)+$count]
                                        = $reg[$op1];
1197                            }
1198                            else{
1199                                    $calc_count = $reg[$op1] % $reg[$op2
                                        ]; ## <-- Remainder
1200                                    $reg[$op1] = $reg[$op1] -
                                        $calc_count;
1201                                    $padding = $reg[$op1] / $reg[$op2];
                                        ## <-- Quotient
1202                                    $reg[$op2] = $calc_count;
1203                                    $reg[$op1] = $padding;
1204                                    if((($neg_check == 1)&&($op1 != $op2
                                        ))||(($neg_check == 2)&&(
                                        $operand1_reg != $operand2_reg)))
                                         {
1205                                            if((($bug1 eq "neg")||(
                                                $neg_check == 2))&&((
                                                $operand2_reg != 1)&&(
                                                $calc_count != 0))) {
1206                                                    $reg[$op1] = $reg[
                                                        $op1] + 1;
1207                                                    $calc_count = ($reg[
                                                        $op1]*
                                                        $operand2_reg) -
                                                        $operand1_reg;
1208                                                    $reg[$op2] =
                                                        $calc_count;
```

```
1209                                                    $bug1 = 0;
1210                                                }
1211                                                if($neg_check == 1){
1212                                                    $twos_comp = $reg[
                                                        $op1];
1213                                                    $twos_size =
                                                        $bus_size;
1214                                                   &complement;
1215                                                    $reg[$op1] = oct("0
                                                        b$twos_comp"); }
1216                                            }
1217                                            $op_reg[($op2*$memory_size)+$count]
                                                    = $reg[$op2];
1218                                            $op_reg[($op1*$memory_size)+$count]
                                                    = $reg[$op1];
1219                                    }
1220                                    if($reg[$op1] == 0){ $sf[0] = 1; } ## Zero
                                        Flag
1221                                    else { $sf[0] = 0; }
1222                                    $neg_check = 0;
1223                                    $bug1 = sprintf("%X", $reg[$op1]);
1224                                    print $debug_handle "Quo:$op1 \[$bug1\]\t";
1225                                    $bug2 = sprintf("%X", $reg[$op2]);
1226                                    print $debug_handle "Rem:$op2 \[$bug2\]\n\n"
                                        ;
1227                            }
1228                    }
1229                    elsif($instruction eq "DIVC"){
1230                            if($op2 == 0) { while ($op2 < 1) { $op2 = int(rand(
                                $num_reg-1));}}
1231                            $neg_check = 0;
1232                            $bug1 = sprintf("%X", $reg[$op1]);
1233                            $bug2 = sprintf("%X", $op2);
1234                            print  $debug_handle "$op1 \[$bug1\]\t$op2 \[$bug2
                                \]\t";
1235                            if($reg[$op1]>= $twos_test){
1236                                    ++$neg_check;
1237                                    $twos_comp = $reg[$op1];
1238                                    $twos_size = $bus_size;
1239                                   &complement;
1240                                    $reg[$op1] = oct("0b$twos_comp");
1241                                    $bug1 = "neg";
1242                            }
1243                            if(($op2 > $reg[$op1])&&($reg[$op1] != 0)){
1244                                    $calc_count = (1 x $bus_size);
1245                                    $calc_count = oct("0b$calc_count");
1246                                    $reg[$op1] = ($bug1 ne "neg") ? 0 :
                                        $calc_count;
1247                                    $bug1 = 0;
```

```
1248                    }
1249                    else{
1250                            $calc_count = $reg[$op1] % $op2;
1251                            $reg[$op1] = $reg[$op1] - $calc_count;
1252                            $reg[$op1] = $reg[$op1] / $op2;
1253                            if($neg_check == 1){
1254                                    $reg[$op1] = ($calc_count == 0)?
                                           $reg[$op1]: ++$reg[$op1];
1255                                    $twos_comp = $reg[$op1];
1256                                    $twos_size = $bus_size;
1257                                    &complement;
1258                                    $reg[$op1] = oct("0b$twos_comp");
1259                            }
1260                    }
1261                    $neg_check = 0;
1262                    if($reg[$op1] == 0){ $sf[0] = 1; } ## Zero Flag
1263                    else { $sf[0] = 0; }
1264                    $op_reg[($op1*$memory_size)+$count] = $reg[$op1];
1265                    $bug1 = sprintf("%X", $reg[$op1]);
1266                    print $debug_handle "Quo:$op1 \[$bug1\]\n\n";
1267            }
1268            elsif($instruction eq "NOT"){
1269                    $bug1 = sprintf("%X", $reg[$op1]);
1270                    print $debug_handle "$op1 \[$bug1\]\t";
1271                    $calc_count = sprintf("%b", $reg[$op1]);
1272                    $length = length($calc_count);
1273                    $padding = $padding = (0 x (${bus_size} - $length));
1274                    $calc_count = $padding.$calc_count;
1275                    @bin = split(//, $calc_count);
1276                    for my $nott (@bin) {
1277                            if ($nott == 1) { $nott =0;} else {$nott =
                               1;}
1278                    }
1279                    $reg[$op1] = join('',@bin);
1280                    $reg[$op1] = oct("0b".$reg[$op1]);
1281                    &compactor;
1282                    if($reg[$op1] == 0){ $sf[0] = 1; } else { $sf[0] =
                           0; } ## Zero Flag
1283                    $op_reg[($op1*$memory_size)+$count] = $reg[$op1];
1284                    $bug1 = sprintf("%X", $reg[$op1]);
1285                    print $debug_handle "Res:$op1 \[$bug1\]\n\n";
1286                    }
1287            elsif($instruction eq "AND"){
1288                    $bug1 = sprintf("%X", $reg[$op1]);
1289                    $bug2 = sprintf("%X", $reg[$op2]);
1290                    print $debug_handle "$op1 \[$bug1\]\t$op2 \[$bug2\]\
                           t";
1291                    $reg[$op1] = $reg[$op1] & $reg[$op2];
1292                    &compactor;
```

```
1293                    if($reg[$op1] == 0){ $sf[0] = 1; } else { $sf[0] =
                            0; } ## Zero Flag
1294                    $op_reg[($op1*$memory_size)+$count] = $reg[$op1];
1295                    $bug1 = sprintf("%X", $reg[$op1]);
1296                    print $debug_handle "Res:$op1 \[$bug1\]\n\n";
1297                    $sf[2] = ($reg[$op1] >= $twos_test) ? 1 : 0; ##
                            Negative Flag
1298                    }
1299                elsif($instruction eq "OR"){
1300                    $bug1 = sprintf("%X", $reg[$op1]);
1301                    $bug2 = sprintf("%X", $reg[$op2]);
1302                    print $debug_handle "$op1 \[$bug1\]\t$op2 \[$bug2\]\
                            t";
1303                    $reg[$op1] = $reg[$op1] | $reg[$op2];
1304                    &compactor;
1305                    if($reg[$op1] == 0){ $sf[0] = 1; } else { $sf[0] =
                            0; } ## Zero Flag
1306                    $op_reg[($op1*$memory_size)+$count] = $reg[$op1];
1307                    $bug1 = sprintf("%X", $reg[$op1]);
1308                    print $debug_handle "Res:$op1 \[$bug1\]\n\n";
1309                    $sf[2] = ($reg[$op1] >= $twos_test) ? 1 : 0; ##
                            Negative Flag
1310                    }
1311                elsif($instruction eq "XOR"){
1312                    $bug1 = sprintf("%X", $reg[$op1]);
1313                    $bug2 = sprintf("%X", $reg[$op2]);
1314                    print $debug_handle "$op1 \[$bug1\]\t$op2 \[$bug2\]\
                            t";
1315                    $reg[$op1] = $reg[$op1] ^ $reg[$op2];
1316                    &compactor;
1317                    if($reg[$op1] == 0){ $sf[0] = 1; } else { $sf[0] =
                            0; } ## Zero Flag
1318                    $op_reg[($op1*$memory_size)+$count] = $reg[$op1];
1319                    $bug1 = sprintf("%X", $reg[$op1]);
1320                    print $debug_handle "Res:$op1 \[$bug1\]\n\n";
1321                    $sf[2] = ($reg[$op1] >= $twos_test) ? 1 : 0; ##
                            Negative Flag
1322                    }
1323                elsif(($instruction eq "SHLL") or ($instruction eq "SHLA")){
1324                    $bug1 = sprintf("%X", $reg[$op1]);
1325                    $bug2 = sprintf("%X", $op2);
1326                    print $debug_handle "$op1 \[$bug1\]\t$op2 \[$bug2\]\
                            t";
1327                    $reg[$op1] = ($reg[$op1] << $op2);
1328
1329                    &compactor;
1330                    if($reg[$op1] == 0){ $sf[0] = 1; } else { $sf[0] =
                            0; } ## Zero Flag
1331                    $op_reg[($op1*$memory_size)+$count] = $reg[$op1];
```

```
1332                    $bug1 = sprintf("%X", $reg[$op1]);
1333                    print $debug_handle "Res:$op1 \[$bug1\]\n\n";
1334                    }
1335                elsif($instruction eq "SHRL"){
1336                    $bug1 = sprintf("%X", $reg[$op1]);
1337                    $bug2 = sprintf("%X", $op2);
1338                    print $debug_handle "$op1 \[$bug1\]\t$op2 \[$bug2\]\
                        t";
1339                    $reg[$op1] = ($reg[$op1] >> $op2);
1340                    &compactor;
1341                    if($reg[$op1] == 0){ $sf[0] = 1; } else { $sf[0] =
                        0; } ## Zero Flag
1342                    $op_reg[($op1*$memory_size)+$count] = $reg[$op1];
1343                    $bug1 = sprintf("%X", $reg[$op1]);
1344                    print $debug_handle "Res:$op1 \[$bug1\]\n\n";
1345                    }
1346                elsif($instruction eq "SHRA"){
1347                    if($op2 > ((2**${op2_size})-1))
1348                            { print"Warning: Opearnd 2 size exceeding
                                length at no $count : $instruction\n";};
1349                    ##--decimal to binary (in array) conversion --
1350                    $bug1 = sprintf("%X", $reg[$op1]);
1351                    $bug2 = sprintf("%X", $op2);
1352                    print $debug_handle "$op1 \[$bug1\]\t$op2 \[$bug2\]\
                        t";
1353                    $calc_count = sprintf("%b", $reg[$op1]);
1354                    $length = length($calc_count);
1355                    $padding = (0 x (${bus_size} - $length));
1356                    $calc_count = $padding.$calc_count;
1357                    @bin = split(//, $calc_count);
1358                    #-- END --
1359                    @bin = reverse(@bin);
1360                    $padding = ((@bin[${bus_size}-1]) x ${bus_size});
1361                    $calc_count = $padding.$calc_count;
1362                    @bin = split(//, $calc_count);
1363                    @bin = reverse(@bin);
1364                    @temp_array = @bin[$op2 .. ($op2+${bus_size}-1)];
1365                    @temp_array = reverse(@temp_array);
1366                    $calc_count = join('',@temp_array);
1367                    $calc_count = oct("0b$calc_count");
1368                    $reg[$op1] = ${calc_count};
1369                    &compactor;
1370                    if($reg[$op1] == 0){ $sf[0] = 1; } else { $sf[0] =
                        0; } ## Zero Flag
1371                    $op_reg[($op1*$memory_size)+$count] = $reg[$op1];
1372                    $bug1 = sprintf("%X", $reg[$op1]);
1373                    print $debug_handle "Res:$op1 \[$bug1\]\n\n";
1374                    }
1375                elsif($instruction eq "ROTL"){
```

```
1376                    if($op2 > ((2**${op2_size})-1))
1377                            { print"Warning: Opearnd 2 size exceeding
                                   length at no $count : $instruction\n";};
1378                    ##--decimal to binary (in array) conversion --
1379                    $bug1 = sprintf("%X", $reg[$op1]);
1380                    $bug2 = sprintf("%X", $op2);
1381                    print $debug_handle "$op1 \[$bug1\]\t$op2 \[$bug2\]\
                           t";
1382                    $calc_count = sprintf("%b", $reg[$op1]);
1383                    $length = length($calc_count);
1384                    $padding = (0 x (${bus_size} - $length));
1385                    $calc_count = $padding.$calc_count;
1386                    @bin = split(//, $calc_count);
1387                    #-- END --
1388                    @bin = (@bin,@bin);
1389                    @temp_array = @bin[$op2 .. (($op2+${bus_size}-1))];
1390                    $calc_count = join('',@temp_array);
1391                    $calc_count = oct("0b$calc_count");
1392                    $reg[$op1] = $calc_count;
1393                    if($reg[$op1] == 0){ $sf[0] = 1; } else { $sf[0] =
                           0; } ## Zero Flag
1394                    $op_reg[($op1*$memory_size)+$count] = $reg[$op1];
1395                    $bug1 = sprintf("%X", $reg[$op1]);
1396                    print $debug_handle "Res:$op1 \[$bug1\]\n\n";
1397                    }
1398            elsif($instruction eq "ROTR"){
1399                    if($op2 > ((2**${op2_size})-1))
1400                            { print"Warning: Opearnd 2 size exceeding
                                   length at no $count : $instruction\n";};
1401                    ##--decimal to binary (in array) conversion --
1402                    $bug1 = sprintf("%X", $reg[$op1]);
1403                    $bug2 = sprintf("%X", $op2);
1404                    print $debug_handle "$op1 \[$bug1\]\t$op2 \[$bug2\]\
                           t";
1405                    $calc_count = sprintf("%b", $reg[$op1]);
1406                    $length = length($calc_count);
1407                    $padding = (0 x (${bus_size} - $length));
1408                    $calc_count = $padding.$calc_count;
1409                    @bin = split(//, $calc_count);
1410                    #-- END --
1411                    @bin = (@bin,@bin);
1412                    @bin = reverse(@bin);
1413                    @temp_array = @bin[$op2 .. (($op2+${bus_size}-1))];
1414                    @temp_array = reverse(@temp_array);
1415                    $calc_count = join('',@temp_array);
1416                    $calc_count = oct("0b$calc_count");
1417                    $reg[$op1] = $calc_count;
1418                    if($reg[$op1] == 0){ $sf[0] = 1; } else { $sf[0] =
                           0; } ## Zero Flag
```

```
1419                    $op_reg[($op1*$memory_size)+$count] = $reg[$op1];
1420                    $bug1 = sprintf("%X", $reg[$op1]);
1421                    print $debug_handle "Res:$op1 \[$bug1\]\n\n";
1422                    }
1423            elsif($instruction eq 'RTLC'){
1424                    if($op2 > ((2**${op2_size})-1))
1425                            { print"Warning: Opearnd 2 size exceeding
                                  length at no $count : $instruction\n";};
1426                    ##--decimal to binary (in array) conversion --
1427                    $bug1 = sprintf("%X", $reg[$op1]);
1428                    $bug2 = sprintf("%X", $op2);
1429                    print $debug_handle "$op1 \[$bug1\]\t$op2 \[$bug2\]\
                        t";
1430                    $calc_count = sprintf("%b", $reg[$op1]);
1431                    $length = length($calc_count);
1432                    $padding = (0 x (${bus_size} - $length));
1433                    $calc_count = $padding.$calc_count;
1434                    @bin = split(//, $calc_count);
1435                    #-- END --
1436                    @bin = (@bin,$sf[3],@bin);
1437                    $sf[3] = @bin[(${bus_size}+$op2)];
1438                    @temp_array = @bin[$op2 .. (($op2+${bus_size}-1))];
1439                    $calc_count = join('',@temp_array);
1440                    $calc_count = oct("0b$calc_count");
1441                    $reg[$op1] = $calc_count;
1442                    if($reg[$op1] == 0){ $sf[0] = 1; } else { $sf[0] =
                        0; } ## Zero Flag
1443                    $op_reg[($op1*$memory_size)+$count] = $reg[$op1];
1444                    $bug1 = sprintf("%X", $reg[$op1]);
1445                    print $debug_handle "Res:$op1 \[$bug1\]\n\n";
1446                    }
1447            elsif($instruction eq 'RTRC'){
1448                    if($op2 > ((2**${op2_size})-1))
1449                            { print"Warning: Opearnd 2 size exceeding
                                  length at no $count : $instruction\n";};
1450                    ##--decimal to binary (in array) conversion --
1451                    $bug1 = sprintf("%X", $reg[$op1]);
1452                    $bug2 = sprintf("%X", $op2);
1453                    print $debug_handle "$op1 \[$bug1\]\t$op2 \[$bug2\]\
                        t";
1454                    $calc_count = sprintf("%b", $reg[$op1]);
1455                    $length = length($calc_count);
1456                    $padding = (0 x (${bus_size} - $length));
1457                    $calc_count = $padding.$calc_count;
1458                    @bin = split(//, $calc_count);
1459                    #-- END --
1460                    @bin = (@bin,$sf[3],@bin);
1461                    @bin = reverse(@bin);
1462                    $sf[3] = @bin[(${bus_size}+$op2)];
```

```
1463                    @temp_array = @bin[$op2 .. (($op2+${bus_size}-1))];
1464                    @temp_array = reverse(@temp_array);
1465                    $calc_count = join('',@temp_array);
1466                    $calc_count = oct("0b$calc_count");
1467                    $reg[$op1] = $calc_count;
1468                    if($reg[$op1] == 0){ $sf[0] = 1; } else { $sf[0] =
                            0; } ## Zero Flag
1469                    $op_reg[($op1*$memory_size)+$count] = $reg[$op1];
1470                    $bug1 = sprintf("%X", $reg[$op1]);
1471                    print $debug_handle "Res:$op1 \[$bug1\]\n\n";
1472                    }
1473                elsif($instruction eq "COPY"){
1474                    $bug1 = sprintf("%X", $reg[$op1]);
1475                    $bug2 = sprintf("%X", $op2);
1476                    print $debug_handle "$op1 \[$bug1\]\t$op2 \[$bug2\]\
                            t";
1477                    $reg[$op1] = $reg[$op2];
1478                    $op_reg[($op1*$memory_size)+$count] = $reg[$op1];
1479                    $bug1 = sprintf("%X", $reg[$op1]);
1480                    print $debug_handle "Res1:$op1 \[$bug1\]\t";
1481                    $bug2 = sprintf("%X", $reg[$op2]);
1482                    print $debug_handle "Res2:$op2 \[$bug2\]\n\n";
1483                    }
1484                elsif($instruction eq "SWAP"){
1485                    $bug1 = sprintf("%X", $reg[$op1]);
1486                    $bug2 = sprintf("%X", $op2);
1487                    print $debug_handle "$op1 \[$bug1\]\t$op2 \[$bug2\]\
                            t";
1488                    $calc_count = $reg[$op1];
1489                    $reg[$op1] = $reg[$op2];
1490                    $op_reg[($op1*$memory_size)+$count] = $reg[$op1];
1491                    $reg[$op2] = $calc_count;
1492                    $op_reg[($op2*$memory_size)+$count] = $reg[$op2];
1493                    $bug1 = sprintf("%X", $reg[$op1]);
1494                    print $debug_handle "Res1:$op1 \[$bug1\]\t";
1495                    $bug2 = sprintf("%X", $reg[$op2]);
1496                    print $debug_handle "Res2:$op2 \[$bug2\]\n\n";
1497                    }
1498                ## value calculation end
1499                #putting binary together
1500                if($shize == 1) { $shize  = 0; }
1501                else {
1502                    @bin = reverse(@sf);
1503                    $calc_count = join('', @bin);
1504                    if($bus_size == 14)
1505                    { $padding = (0 x 4); }
1506                    else {$padding = (0 x 8); }
1507                    $calc_count = $calc_count.$padding;
1508                    $calc_count = oct("0b$calc_count");
```

```
1509                            $calc_count = sprintf("%X", $calc_count);
1510                            $sf_flag = $calc_count;
1511                            $status_op[$count] = $calc_count;
1512                           &iw1_gen; }
1513 }
1514 ##------- manipulation_gen Ends ---
1515
1516 ##------- iw1_gen -----------------
1517 ## packs instruction word from opcode and operand information
1518 sub iw1_gen{
1519         my $hex1;
1520         $temp_inst = sprintf ("%b", hex(${instruction_no}));
1521         $length = length($temp_inst); ## find lenth of binary generated to
                find 0's padding
1522         $padding = (0 x (${opcode_size} - $length));
1523         $temp_inst = $padding.$temp_inst;
1524         $temp_op1 = sprintf ("%b", ${op1});
1525         $length = length($temp_op1);
1526         if(($bus_size == 12)&&($flow == 1)){
1527                 $padding = (0 x (($op1_size - 1) - $length));
1528         }
1529         else{
1530                 $padding = (0 x (${op1_size} - $length));
1531         }
1532         $temp_op1 = $padding.$temp_op1;
1533         $temp_op2 = sprintf ("%b", ${op2});
1534         $length = length($temp_op2);
1535         $padding = (0 x (${op2_size} - $length));
1536         $temp_op2 = $padding.$temp_op2;
1537         if($flow == 1){
1538                 $IW = $temp_inst.$temp_op1.$condition;
1539         }
1540         else{
1541                 $IW = $temp_inst.$temp_op1.$temp_op2;
1542         }
1543         $IW = sprintf ("%X", oct("0b${IW}"));
1544         $pm[$count] = $IW;
1545         $pm_out[$count] = hex($IW);
1546         $inst[$count] = ($jump == 1)? ('*'.$instruction): (($call != 0)? ('
                **'.$instruction): $instruction);
1547         $operand1[$count] = $op1;
1548         $operand2[$count] = ($flow == 1)? "" : $op2;
1549
1550         if(($jump == 0)||(($jump == 1)&&($bck_jmp > 0))||($IW1_flag == 1)){
                ##condition to skip printing when instructions are jumped over
1551                 $hex1 = sprintf("%X", $count);
1552                 $srno_array[$sr_no] = $hex1;
1553                 ++$sr_no;
1554         }
```

```
1555            ++$count;
1556            if(($jump == 0)||(($jump == 1)&&($bck_jmp > 0))||($IW1_flag == 1)){
1557                    &reg_store;
1558                    print $debug_handle1 "  $count jump: $jump ,, bck_jmp:
                            $bck_jmp \n";
1559            }
1560            else{
1561                    $bck_array[5] = $bck_array[5] + 1;
1562            }
1563            $counter_call = ($call > 0)? $counter_call : ($counter_call + 1);
1564            if($IW1_flag ==1){
1565                    $IW1_flag = 0;
1566                    $address_off = sprintf ("%X",${address_off});
1567                    $pm[$count] = $address_off;
1568                    $pm_out[$count] = hex($address_off);
1569                    $inst[$count] = "";#$instruction;
1570                    $operand1[$count] = "";#" Offset";
1571                    $operand2[$count] = "";
1572
1573                    $hex1 = sprintf("%X", $count);
1574                    $srno_array[$sr_no] = $hex1;
1575                    ++$sr_no;
1576
1577                    ++$count;
1578                    $counter_call = ($call > 0)? $counter_call : ($counter_call
                            + 1);
1579            }
1580            if(($call > 0)&& ($count == $routine_end)){
1581                    $instruction_no = $branch{RET};
1582                    $temp_inst = sprintf ("%b", hex(${instruction_no}));
1583                    $length = length($temp_inst); ## find lenth of binary
                            generated to find 0's padding
1584                    $padding = (0 x (${opcode_size} - $length));
1585                    $temp_inst = $padding.$temp_inst;
1586                    $IW = $temp_inst.$temp_op1.$temp_op2;
1587                    $IW = sprintf ("%X", oct("0b${IW}"));
1588                    $pm[$count] = $IW;
1589                    $pm_out[$count] = hex($IW);
1590                    $inst[$count] = "RETURN";#$instruction;
1591                    $operand1[$count] = "";#" Offset";
1592                    $operand2[$count] = "";
1593
1594                    $hex1 = sprintf("%X", $count); ##printing purpose
1595                    $srno_array[$sr_no] = $hex1;
1596                    ++$sr_no;
1597
1598                    $routine_end = pop @internal_stack;
1599                    $calc_count = pop @internal_stack;
1600                    @sf = split('',$calc_count);
```

```
1601                    @bin = reverse(@sf);
1602                    $calc_count = join('', @bin);
1603                    if($bus_size == 14)
1604                    { $padding = (0 x 4); }
1605                    else {$padding = (0 x 8); }
1606                    $calc_count = $calc_count.$padding;
1607                    $calc_count = oct("0b$calc_count");
1608                    $calc_count = sprintf("%X", $calc_count);
1609                    $sf_flag = $calc_count;
1610                    $count = pop @internal_stack;
1611                    $call = $call - 1;
1612                    $routine_end = ($call > 0) ? (pop @internal_stack) :
                            $routine_end;
1613                    if($call > 0) { push @internal_stack, $routine_end; }
1614                    $SP = ${sp_direct} ? (${SP} + 2):(${SP} - 2);
1615                    $instruction = "RET";
1616                   &reg_store;
1617            }
1618            $flow = 0;
1619            if(((($bck_jmp == 1)&&($jump == 1))||(($bck_jmp == 2)&&($count ==
                    $bck_array[3]))||(($bck_jmp == 3)&&($count == $bck_array[1])))){ #
                    #putother or conditio to chane pc on 2nd & 3rd jump
1620                    print $debug_handle1 "\tbck: $bck_jmp\n";
1621                    if(($bck_jmp == 1)&&($jump == 1)){
1622                            $count = $bck_array[1];
1623                            print $debug_handle1 "de0: $bck_jmp ,, count(in ret0
                                    ): $count\n\n";}
1624                    elsif(($bck_jmp == 2)&&($count == $bck_array[3])){
1625                            $count = $bck_array[4];
1626                            print $debug_handle1 "de1: $bck_jmp ,, count(in ret1
                                    ): $count\n\n";}
1627                    else{
1628                            $count = $bck_array[3];
1629                            $bck_jmp = 0;
1630                            $jump = 0;
1631                            print $debug_handle1 "de2: $bck_jmp ,, count(in ret2
                                    ): $count\n\n";}
1632                    $jump = 0;
1633            }
1634    }
1635    ##------- iw1_gen Ends -------
1636
1637    ##------- reg_store ----------
1638    ## ----- Reg_store: store register values in arrays ----
1639    sub reg_store{
1640            $inst_sequence = ($count == 1) ? 0 : $inst_sequence;
1641            if($manip{$instruction}){
1642                    $inst_sequence = ($count == 1) ? 0 : ++$inst_sequence;
1643            }
```

```
1644            elsif($data_transfer{$instruction}){
1645                    $inst_sequence = ($count == 1) ? ++$inst_sequence : (
                            $inst_sequence + 2);
1646            }
1647            else{
1648                    if($instruction eq "RET"){
1649                            ++$inst_sequence;
1650                    }
1651                    else{
1652                            $inst_sequence = ($count == 1) ? ++$inst_sequence :
                                    ($inst_sequence + 2);
1653                    }
1654            }
1655            for my $i (0 .. ($num_reg - 1)){
1656                    $reg_array[($i*$no_inst)+ $inst_sequence] = $reg[$i];
1657            }
1658            $flag_array[$inst_sequence] = $sf_flag;
1659    }
1660    ##------- Reg_store Ends -----
1661
1662    ##------- compactor ----------
1663    ## ----- Compactor: to shorten the result to correct bit size ----
1664    sub compactor{
1665            $calc_count = sprintf("%b", $reg[$op1]);
1666            $length = length($calc_count);
1667            if($length < $bus_size){
1668                    $padding = (0 x ($bus_size - $length));
1669                    $calc_count = $padding.$calc_count;
1670            }
1671            @temp_array = split('', $calc_count);
1672            @temp_array = reverse(@temp_array);
1673            @bin = @temp_array[0 .. ($bus_size - 1)];
1674            @bin = reverse(@bin);
1675            $calc_count = join('', @bin);
1676            $reg[$op1] = oct("0b$calc_count");
1677    }
1678    ##------- compactor Ends -------
1679
1680    ##------- complement ----------
1681    ## 2's complement fucntion
1682    sub complement
1683    {
1684            $twos_comp = sprintf("%b", $twos_comp);
1685            $length = length($twos_comp);
1686            if($length > $twos_size){
1687                    print "Warning: Length exceeding ${twos_size} at instruction
                            number $count : $instruction\n";
1688            }
1689            $padding = (0 x (${twos_size} - $length));
```

```
1690            $twos_comp = $padding.$twos_comp;
1691            @temp_array = split(//, $twos_comp);
1692            @temp_array = reverse (@temp_array);
1693            $temp_flag = 0;
1694            for my $i (0..($twos_size-1)){
1695                    if($temp_flag == 1){
1696                            $temp_array[$i] = $temp_array[$i] ? 0 : 1;
1697                    }
1698                    elsif(($temp_flag == 0)&&($temp_array[$i] == 1)){
1699                            $temp_flag = 1;
1700                    }
1701            }
1702            @temp_array = reverse (@temp_array);
1703            $twos_comp = join('',@temp_array);
1704    }
1705    ##------- complement Ends ---
1706
1707    ##------- CNVZ ---------------
1708    ## CNVZ flag assignment Function
1709    sub CNVZ
1710    {
1711            $calc_count = sprintf("%b", $reg[$op1]);
1712            $length = length($calc_count);
1713            if($length < $bus_size){
1714                    $padding = (0 x ($bus_size - $length));
1715                    $calc_count = $padding.$calc_count;
1716            }
1717            @temp_array = split('', $calc_count);
1718            @temp_array = reverse(@temp_array);
1719            if($length > $bus_size){
1720                    $sf[3] = $temp_array[$bus_size]; }
1721            else { $sf[3] = 0; }
1722            @bin = @temp_array[0 .. ($bus_size - 1)];
1723            @bin = reverse(@bin);
1724            $calc_count = join('', @bin);
1725            $calc_count = oct("0b$calc_count");
1726            if($calc_count == 0){ $sf[0] = 1; } else { $sf[0] = 0; } ## Zero
                Flag
1727            if ( $bin[0] == 1){ $sf[2] = 1; } else { $sf[2] = 0; } ## Negative
                Flag
1728            $calc_count = sprintf("%b", $operand1_reg);
1729            $length = length($calc_count);
1730            @temp_array = split("",$calc_count);
1731            @temp_array = reverse(@temp_array);
1732            if($length > ($bus_size-2)){
1733                    $calc_count = $bus_size -1;
1734                    $calc_count = $temp_array[$calc_count];
1735                    $operand1_reg = ($calc_count) ? 1 : 0 ;
1736            }
```

```
1737            else { $operand1_reg = 0; }
1738            $calc_count = sprintf("%b", $operand2_reg);
1739            $length = length($calc_count);
1740            @temp_array = split("",$calc_count);
1741            @temp_array = reverse(@temp_array);
1742            if($length > ($bus_size-2)){
1743                    $calc_count = $bus_size -1;
1744                    $calc_count = $temp_array[$calc_count];
1745                    $operand2_reg = ($calc_count) ? 1 : 0 ;
1746            }
1747            else { $operand2_reg = 0; }
1748            $calc_count = ($operand1_reg == $operand2_reg) ? 1 : 0; #XNOR_1st
1749            $length = $calc_count & $operand1_reg;
1750            $padding = $calc_count & $bin[0];
1751            $calc_count = ($padding == $length) ? 0 : 1; #XOR_final
1752            $sf[1] = $calc_count; ## Over-Flow flag
1753    }
1754    ##-------- CNVZ Ends ---------
1755
1756    ##-------- error_read_out ---
1757    ## Error reporting
1758    sub error_read_out
1759    {
1760            if(${flag} == 1) { die " Mnemonic mapping should be closed before
                    line no.$file_line in file:$config \n"; }
1761            elsif(${flag} == 2) { die " Manipulation instructions should be
                    closed before line no.$file_line in file:$config \n"; }
1762            elsif(${flag} == 3) { die " Branch instructions should be closed
                    before line no.$file_line in file:$config \n"; }
1763            elsif(${flag} == 4) { die " Data Transfer instructions should be
                    closed before line no.$file_line in file:$config \n"; }
1764    }
1765    ##-------- error_read_out Ends --
```

III.2 Extract function

```
1
2   sub extract($, $, $)
3   {
4           my $temp_1; my $temp_2;
5           $temp_1 = $_[0];
6           $temp_2 = $_[1];
7           $flag = $_[2];
8           if(${temp_1} eq "bits")
9           {
```

```
10                  if(${flag} != 10 && ${flag} !=0) { &error_read_out; }
11                  ${bus_size} = hex(${temp_2});
12                  print "bits: ${bus_size}\n"; ${oth_flag} = 6;    }
13          elsif (${temp_1} eq "registers")
14          {
15                  if(${flag} != 10 && ${flag} !=0) { &error_read_out; }
16                  ${num_reg} = hex(${temp_2});
17                  print "registers: ${num_reg}\n"; ${oth_flag} = 6;       }
18          elsif (${temp_1} eq "architecture")
19          {
20                  if(${flag} != 10 && ${flag} !=0) { &error_read_out; }
21                  ${arch} = hex(${temp_2});
22                  print "architecture: ${arch}\n"; ${oth_flag} = 6;       }
23          elsif (${temp_1} eq "opcode_size")
24          {
25                  if(${flag} != 10 && ${flag} !=0) { &error_read_out; }
26                  ${opcode_size} = hex(${temp_2});
27                  print "opcode_size ${opcode_size}\n"; ${oth_flag} = 6;  }
28          elsif (${temp_1} eq "operand1_size")
29          {
30                  if(${flag} != 10 && ${flag} !=0) { &error_read_out; }
31                  ${op1_size} = hex(${temp_2});
32                  print "operand1_size: ${temp_2}\n"; ${oth_flag} = 6;    }
33          elsif (${temp_1} eq "operand2_size")
34          {
35                  if(${flag} != 10 && ${flag} !=0) { &error_read_out; }
36                  ${op2_size} = hex(${temp_2});
37                  print "operand2_size: ${temp_2}\n"; ${oth_flag} = 6;    }
38          elsif (${temp_1} eq "memory_size")
39          {
40                  if(${flag} != 10 && ${flag} !=0) { &error_read_out; }
41                  $custom_mem = 1;
42                  $calc_count = hex(${temp_2});
43                  $calc_count = (2**$calc_count);
44                  $mem_implemented = $calc_count;
45                  print "memory_size: ${temp_2} : ${mem_implemented}\n"; ${
                        oth_flag} = 6;          }
46          elsif (${temp_1} eq "dm_size")
47          {
48                  if(${flag} != 10 && ${flag} !=0) { &error_read_out; }
49                  $custom_datamem = 1;
50                  $calc_count = hex(${temp_2});
51                  $dm_mask = (1 x $calc_count); print "dm_mask1: $dm_mask \n";
52                  $dm_mask = oct("0b$dm_mask"); print "dm_mask2: $dm_mask\n";
53                  $calc_count = (2**$calc_count);
54                  ${datamem_implemented} = $calc_count;
55                  print "Data memory_size: ${temp_2} : ${datamem_implemented}\
                        n"; ${oth_flag} = 6;          }
56          elsif (${temp_1} eq "pc_in_pc_relative")
```

```
57          {
58                  if(${flag} != 10 && ${flag} !=0) { &error_read_out; }
59                  $calc_count = hex(${temp_2});
60                  $now = ($calc_count)? 0 : 1;
61                  print "pc_in_pc_relative: $now\n"; ${oth_flag} = 6;
                                }
62          elsif (${temp_1} eq "Stack_direction")
63          {
64                  if(${flag} != 10 && ${flag} !=0) { &error_read_out; }
65                  ${sp_direct} = hex(${temp_2});
66                  print "Stack direction: ${sp_direct}\n"; ${oth_flag} = 6;
                                }
67          elsif (${temp_1} eq "SP")
68          {
69                  if(${flag} != 10 && ${flag} !=0) { &error_read_out; }
70                  print "Stack Pointer top: ${temp_2}\t";
71                  ${SP} = hex(${temp_2});
72                  ${SP_top} = ${SP};
73                  print ": ${SP}\n"; ${oth_flag} = 6;                 }
74          elsif (${temp_1} eq "Stack_size")
75          {
76                  if(${flag} != 10 && ${flag} !=0) { &error_read_out; }
77                  ${stack_per} = hex(${temp_2});
78                  print "Percentage of memory for Stack: ${stack_per}\n"; ${
                        oth_flag} = 6;          }
79          if ((${temp_1} eq "start_mapping") or (${temp_1} eq "end_mapping")
                or (${flag} == 1))
80          {
81                  if(${temp_1} eq "start_mapping") {
82                          if(${flag} != 10) {
83                                  print "FLAG: $flag \n";
84                                  die " Mnemonic mapping should be mnemonic :
                                        opcode definitions";
85                          }
86                          else { ${flag} = 1;}
87                  }
88                  elsif(${temp_1} eq "end_mapping")
89                  {       ${flag} = 0;    }
90                  elsif((${temp_1} ne "start_mapping")){
91                          $mnemonics{$temp_1} = "${temp_2}"; ##appending "
                                mnemonic" associative array with 'number' key & '
                                opcode' pair
92                          ++$count;       }
93          }
94          if ((${temp_1} eq "start_manipulation") or (${temp_1} eq "
                end_manipulation") or (${flag} == 2))
95          {
96                  if(${temp_1} eq "EOF") { die " End of File reached &
                        maipulation should be closed before EOF; before line no.
```

```
                    $file_line in file:$config \n"; }
97                  if(${temp_1} eq "start_manipulation") {
98                          if(${flag} != 0) {
99                                  if(${flag} == 1) { die " Mnemonic mapping
                                       should be closed before start_maipulation
                                       ; before line no.$file_line in file:
                                       $config \n"; }
100                                 elsif(${flag} == 3) { die " Branch
                                       instructions should be closed before
                                       start_maipulation; before line no.
                                       $file_line in file:$config \n"; }
101                                 elsif(${flag} == 4) { die " Data Transfer
                                       instructions should be closed before
                                       start_maipulation; before line no.
                                       $file_line in file:$config \n"; }
102                                 elsif(${flag} == 6) { die " Manipulation
                                       instructions should be closed before line
                                        no.$file_line in file:$config \n"; }
103                                 elsif(${flag} == 10) { die " Mnemonic
                                       mapping should be done & closed before
                                       start_maipulation; before line no.
                                       $file_line in file:$config \n"; }
104                         }
105                         else { ${flag} = 2;}
106                 }
107                 if(${temp_1} eq "end_manipulation")
108                 {       ${flag} = 0;    }
109                 elsif((${temp_1} ne "start_manipulation")){
110                         if(exists $mnemonics{$temp_1}) {
111                                 $temp_1 = $mnemonics{$temp_1};
112                                 $manip{"${temp_1}"} = "${temp_2}"; ##
                                       appending "manip" associative array with
                                       'opcode' key & value pair
113                                 --$count;
114                         }
115                         else
116                         { die "Mnemonic ${temp_1} at line no. $file_line in
                                  file:$config not mapped \n"; }
117                 }
118         }
119         if ((${temp_1} eq "start_branch") or (${temp_1} eq "end_branch") or(
               ${flag} == 3))
120         {
121                 if(${temp_1} eq "EOF") { die " End of File reached & branch
                       should be closed before EOF; before line no.$file_line in
                        file:$config \n"; }
122                 if(${temp_1} eq "start_branch") {
123                         if(${flag} != 0) {
```

```
124                                 if(${flag} == 1) { die " Mnemonic mapping
                                        should be closed before  start_branch;
                                        before line no.$file_line in file:$config
                                         \n"; }
125                                 elsif(${flag} == 2) { die " Manipulation
                                        instructions should be closed before
                                        start_branch; before line no.$file_line
                                        in file:$config \n"; }
126                                 elsif(${flag} == 4) { die " Data Transfer
                                        instructions should be closed before
                                        start_branch; before line no.$file_line
                                        in file:$config \n"; }
127                                 elsif(${flag} == 6) { die " Branch
                                        instructions should be closed before line
                                         no.$file_line in file:$config \n"; }
128                                 elsif(${flag} == 10) { die " Mnemonic
                                        mapping should be done & closed before
                                        start_branch; before line no.$file_line
                                        in file:$config \n"; }
129                         }
130                         else { ${flag} = 3;}
131                 }
132                 if(${temp_1} eq "end_branch")
133                 {       ${flag} = 0;    }
134                 elsif(${temp_1} ne "start_branch"){
135                         if(exists $mnemonics{$temp_1}) {
136                                 $temp_1 = $mnemonics{$temp_1};
137                                 $branch{"${temp_1}"} = "${temp_2}"; ##
                                        appending "branch" associative array with
                                         'opcode' key & value pair
138                                 --$count; }
139                         else
140                         { die "Mnemonic ${temp_1} at line no. $file_line in
                                file:$config not mapped \n"; }
141                 }
142         }
143         if ((${temp_1} eq "start_data_transfer") or (${temp_1} eq "
                end_data_transfer") or(${flag} ==  4)) {
144                 if(${temp_1} eq "EOF") { die " End of File reached &
                        data_transfer should be closed before EOF; before line no
                        .$file_line in file:$config \n"; }
145                 if(${temp_1} eq "start_data_transfer") {
146                         if(${flag} != 0) {
147                                 if(${flag} == 1) { die " Mnemonic mapping
                                        should be closed before
                                        start_data_transfer; before line no.
                                        $file_line in file:$config \n"; }
148                                 elsif(${flag} == 2) { die " Manipulation
                                        instructions should be closed before
```

```
                                        start_data_transfer; before line no.
                                        $file_line in file:$config \n"; }
149                                 elsif(${flag} == 3) { die " Branch
                                        instructions should be closed before
                                        start_data_transfer; before line no.
                                        $file_line in file:$config \n"; }
150                                 elsif((${temp_1} ne "start_data_transfer")
                                        and ${flag} == 6) { die " Data Transfer
                                        instructions should be closed before line
                                         no.$file_line in file:$config \n"; }
151                                 elsif(${flag} == 10) { die " Mnemonic
                                        mapping should be done & closed before
                                        start_branch; before line no.$file_line
                                        in file:$config \n"; }
152                             }
153                             else { ${flag} = 4;}
154                     }
155                     if(${temp_1} eq "end_data_transfer")
156                     {       ${flag} = 0;    }
157                     elsif((${temp_1} ne "start_data_transfer")){
158                             if(exists $mnemonics{$temp_1}) {
159                                     $temp_1 = $mnemonics{$temp_1};
160                                     $data_transfer{"${temp_1}"} = "${temp_2}"; #
                                        #appending "data_transfer" associative
                                        array with 'opcode' key & value pair
161                                     --$count; }
162                             else
163                             { die "Mnemonic ${temp_1} at line no. $file_line in
                                    file:$config not mapped \n"; }
164                     }
165             }
166             return ($flag);
167     }
168     1;
```

Appendix IV

Matlab Source Code

Matlab is used to generate graphs in this work. The 'csv' file for a test run is imported into the Matlab and graphs are generated using the following scripts.

IV.1 Errors for tests-Graph1

```
set(0,'defaultAxesFontName', 'Arial')
set(0,'defaultTextFontName', 'Arial')
ab = [Undefinedreg,ImplementationError,ProgramCounter,StatusFlag];
abc = {'Undefinedreg','ImplementationError','ProgramCounter','StatusFlag'};
width = 0.6;
h = bar3(ab,width)
ax = gca;
xtickangle(-45)
xticks(1:1:4)
xticklabels(abc)
ytickangle(19)
yticks(1:1:27)
yticklabels(Instruction);
zlabel('No. of Errors')
cm = get(gcf,'colormap');  % Use the current colormap.
cnt = 0;
for jj = 1:length(h)
    xd = get(h(jj),'xdata');
    yd = get(h(jj),'ydata');
    zd = get(h(jj),'zdata');
```

```
    delete(h(jj))
    idx = [0;find(all(isnan(xd),2))];
    if jj == 1
        S = zeros(length(h)*(length(idx)-1),1);
        dv = floor(size(cm,1)/length(S));
    end
    for ii = 1:length(idx)-1
        cnt = cnt + 1;
        S(cnt) = surface(xd(idx(ii)+1:idx(ii+1)-1,:),...
                         yd(idx(ii)+1:idx(ii+1)-1,:),...
                         zd(idx(ii)+1:idx(ii+1)-1,:),...
                         'facecolor',cm((cnt-1)*dv+1,:));
    end
end
rotate3d
r1 = S(1:27);
r2 = S(28:54);
r3 = S(55:81);
r4 = S(82:108);
set(r1,'facecolor',[0.98 0 0])
set(r2,'facecolor',[1.0 1.0 0.501])
set(r3,'facecolor',[1.0 0.627 0.258])
set(r4,'facecolor',[0.501 1.0 0.501])
```

IV.2 Total Error count-Graph2

```
set(0,'defaultAxesFontName', 'Arial')
set(0,'defaultTextFontName', 'Arial')
width1 = 0.5;
barh(TotalOcc,width1,'FaceColor',[1 1 0.55])
width2 = .25;
hold on
barh(Errors,width2,'FaceColor',[0 0 1])
hold off
ytickangle(0)
yticks(1:1:27)
yticklabels(Instruction)
ylabel('Instructions')
grid on
xlabel('Instruction Count')
legend({'Total Occurance','Errors'},'Location','northwest');
```

Appendix V

Simulation graphs

V.1 Processor *axt*

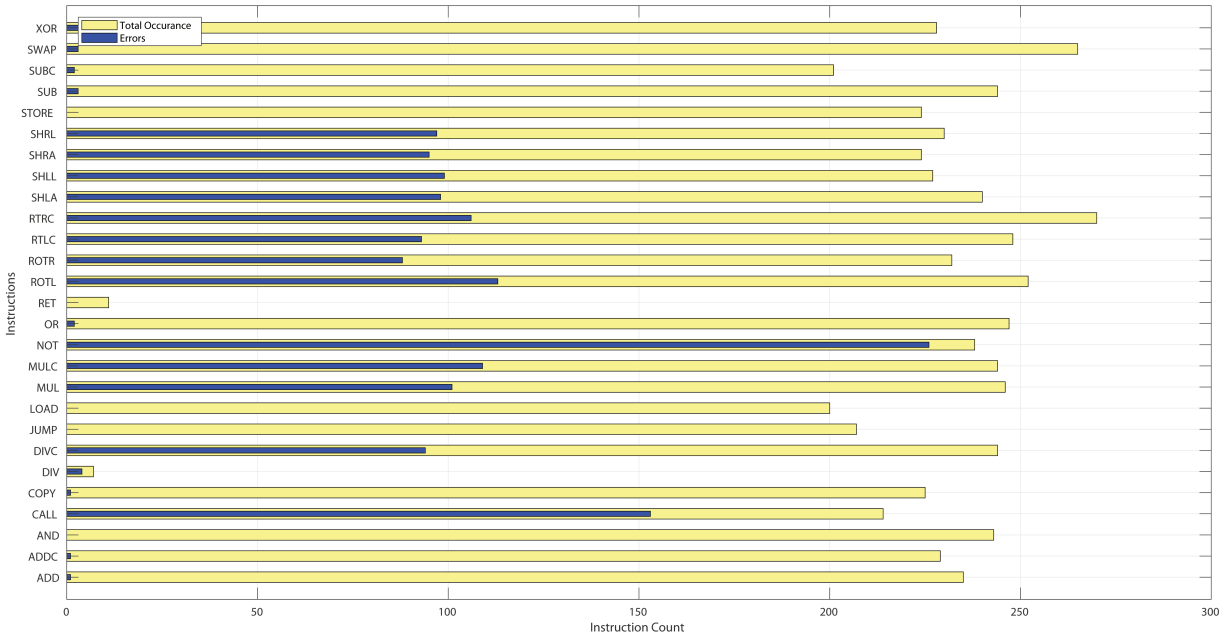

Figure V.1: Total Error count for test *T1* (mode A) in processor *axt*

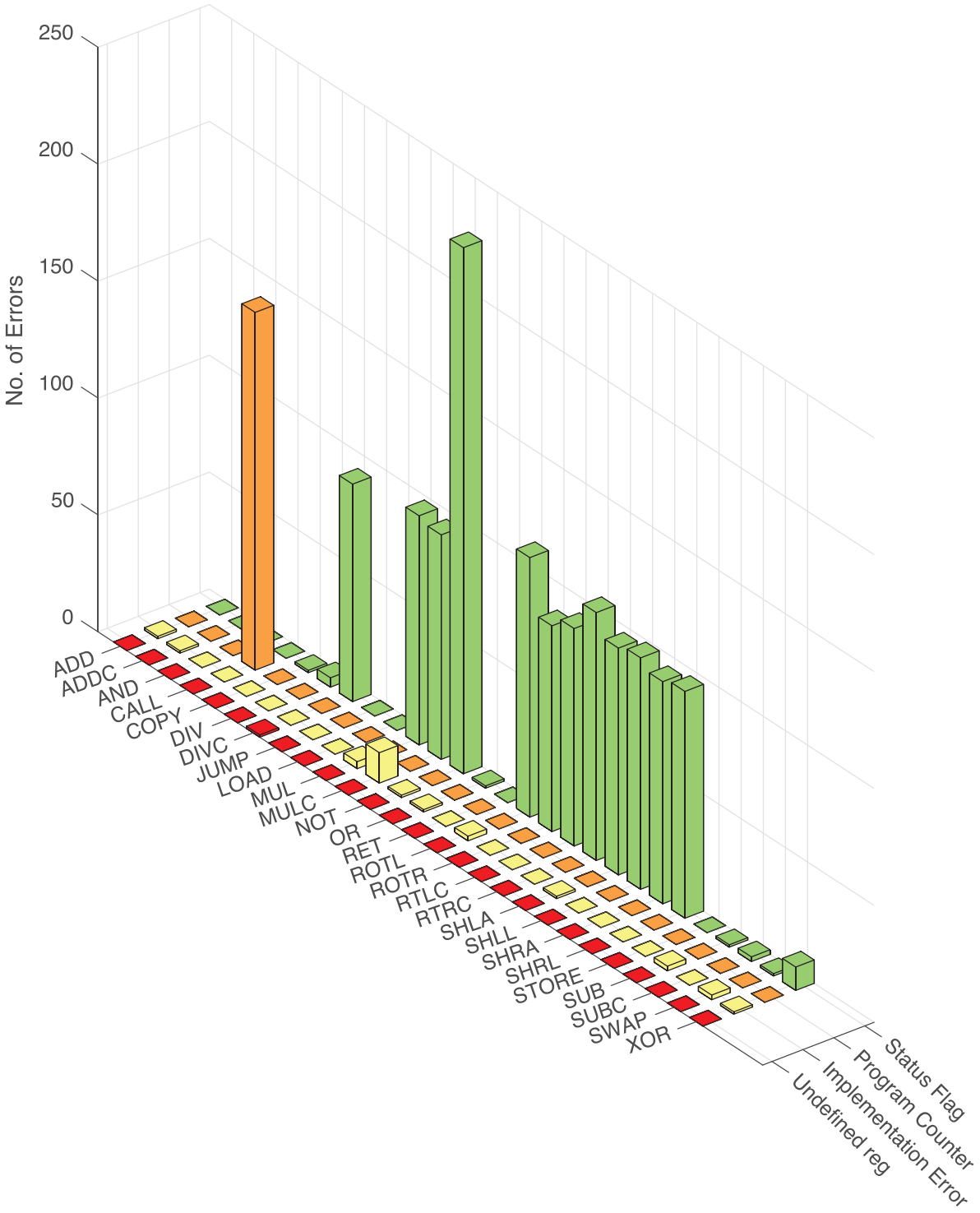

Figure V.2: Errors found in processor *axt* while executing test *T1* (mode A)

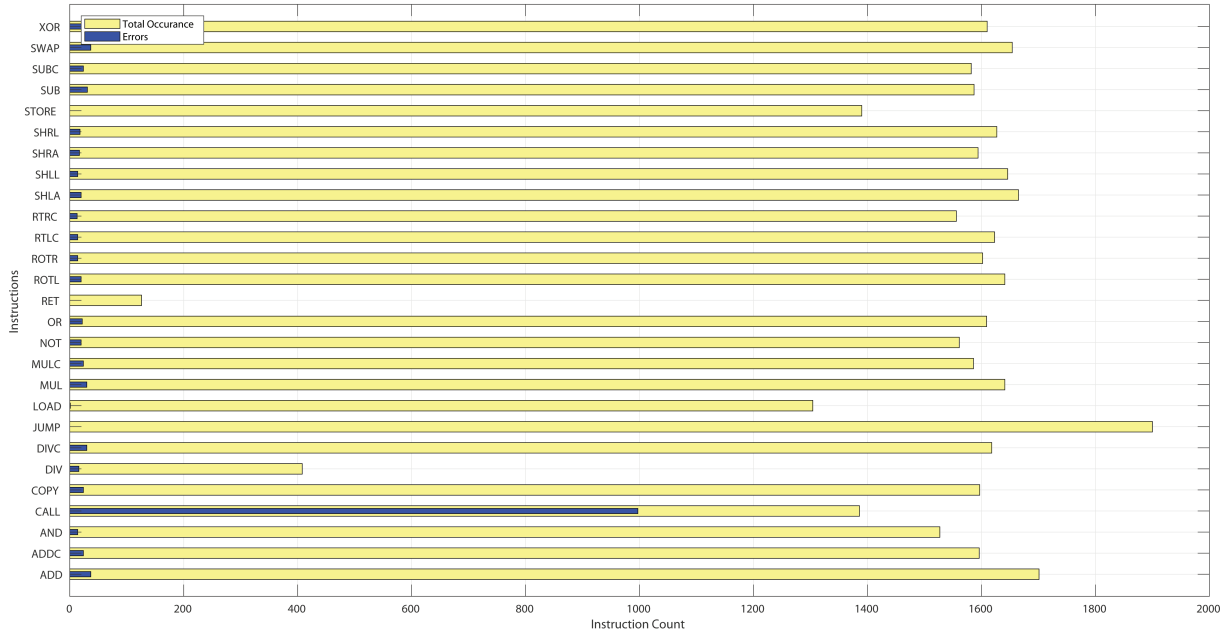

Figure V.3: Total Error count for test *T2* (mode A) in processor *axt*

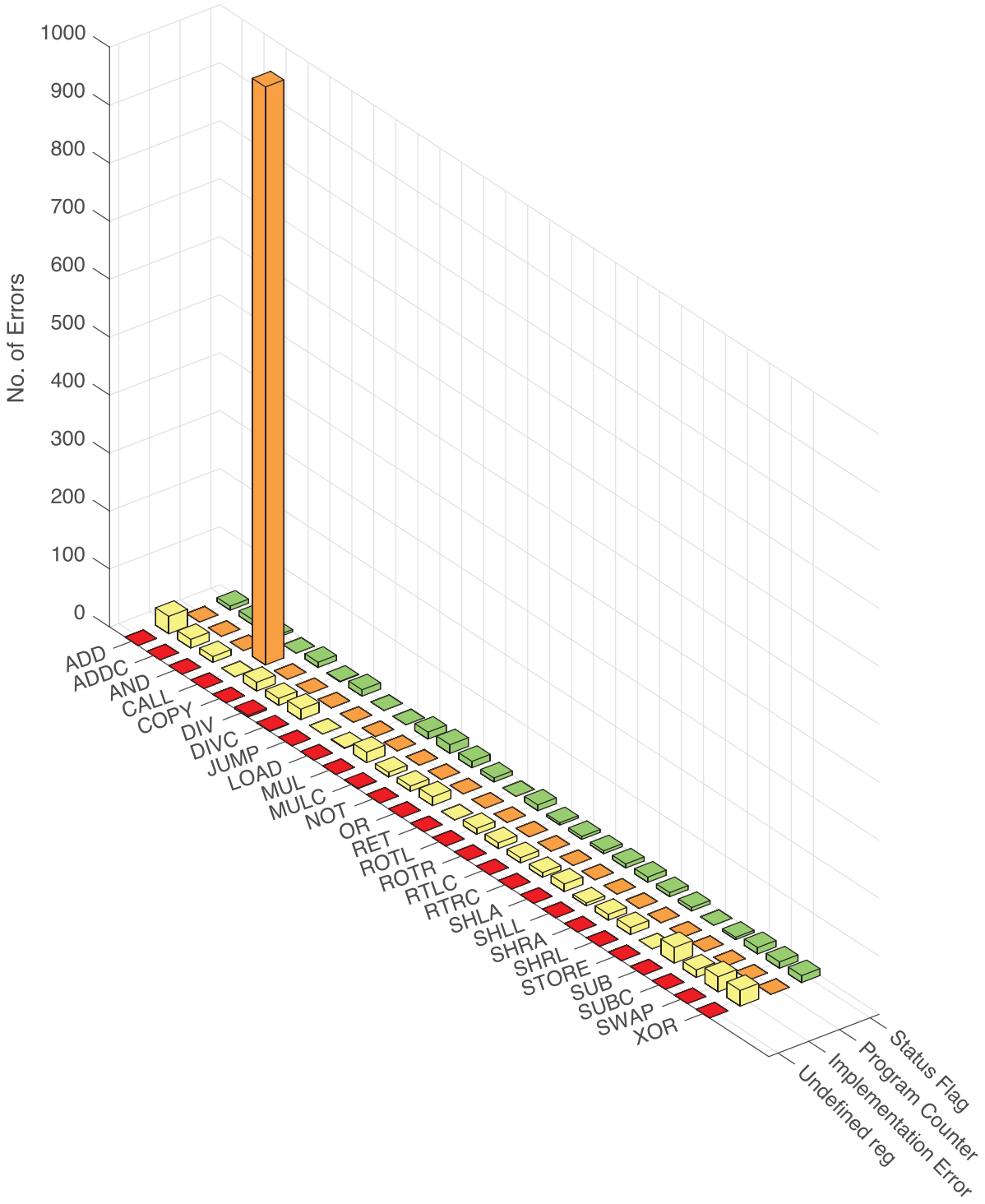

Figure V.4: Errors found in processor *axt* while executing test *T2* (mode A)

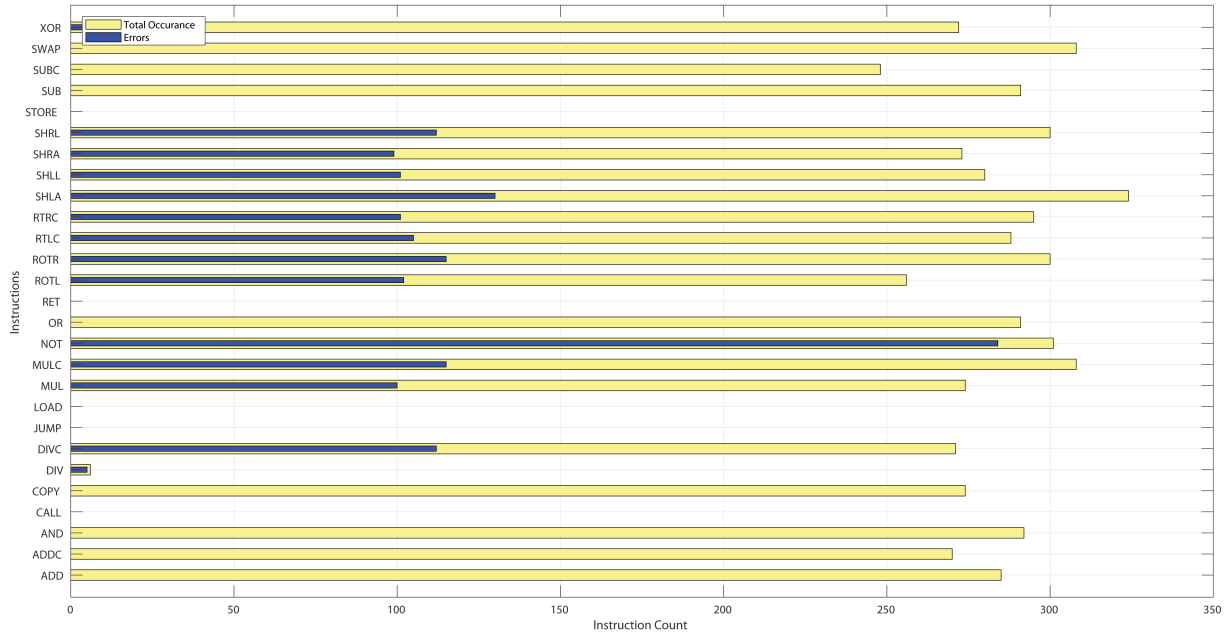

Figure V.5: Total Error count for test *T1* (mode M) in processor *axt*

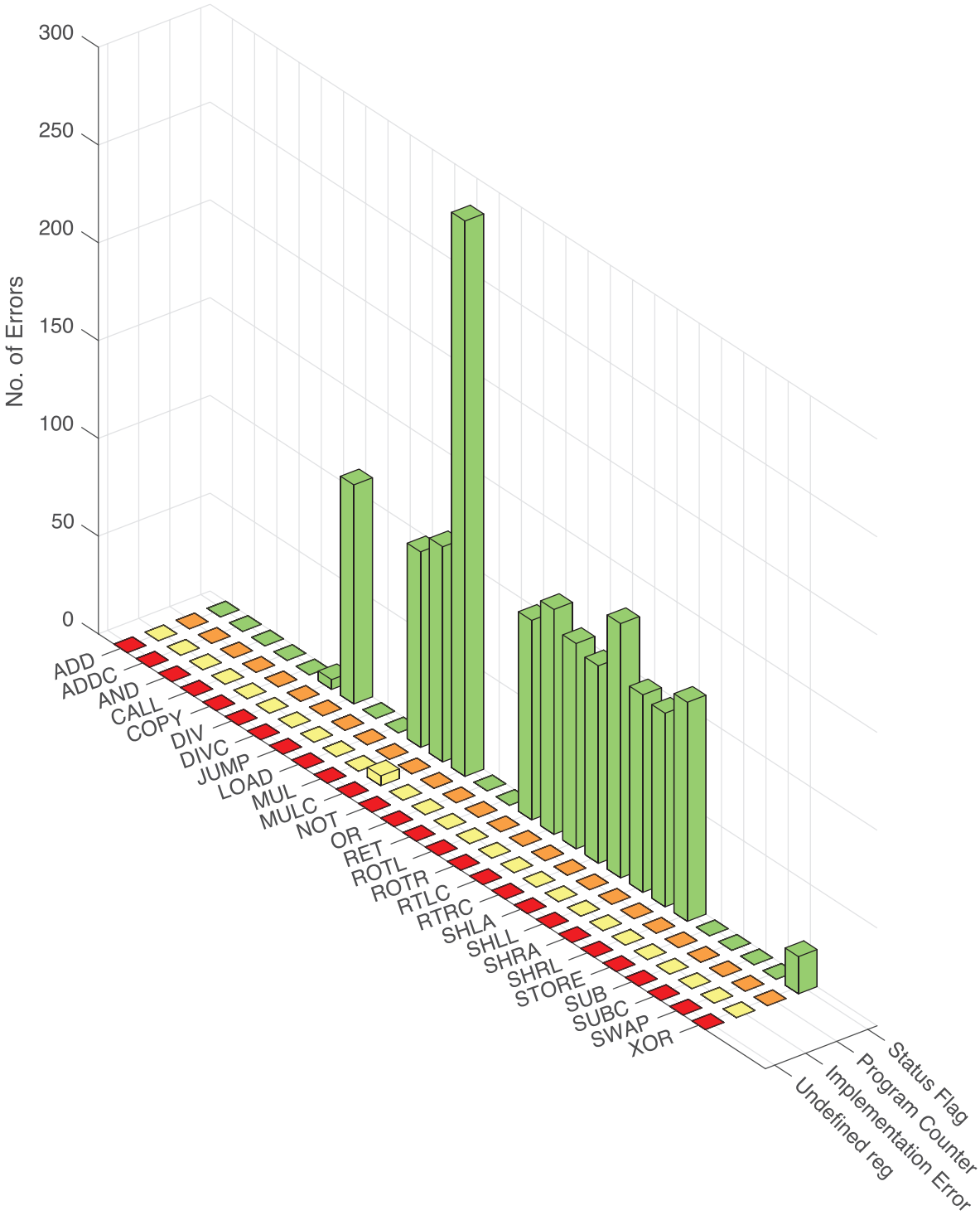

Figure V.6: Errors found in processor *axt* while executing test *T1* (mode M)

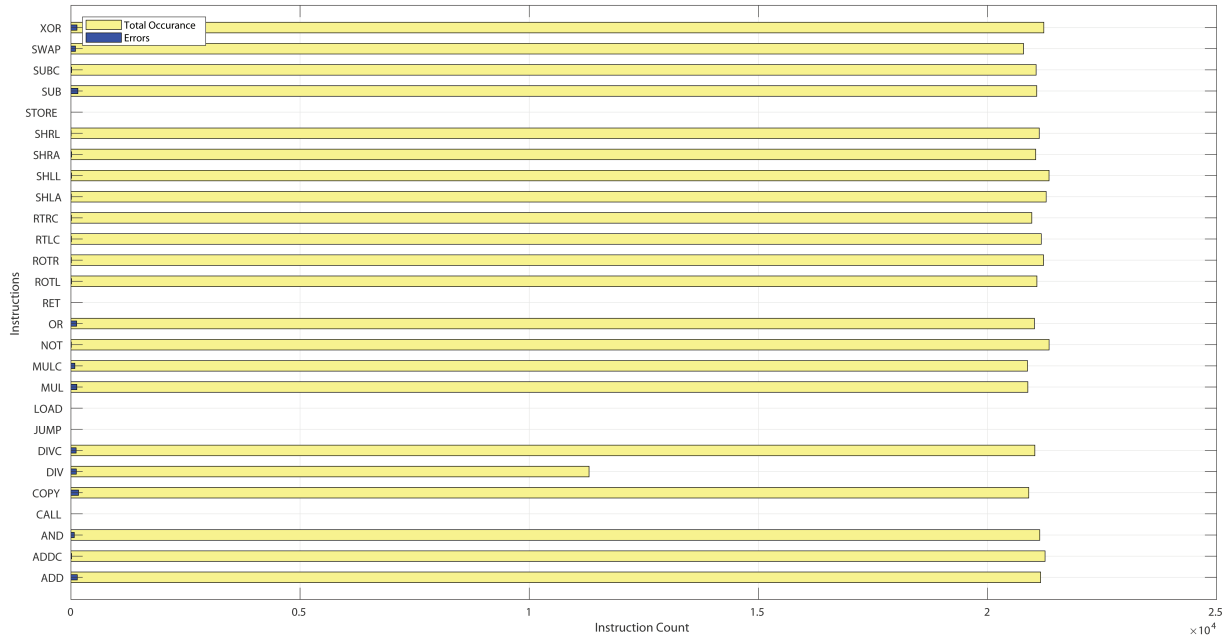

Figure V.7: Total Error count for test *T2* (mode M) in processor *axt*

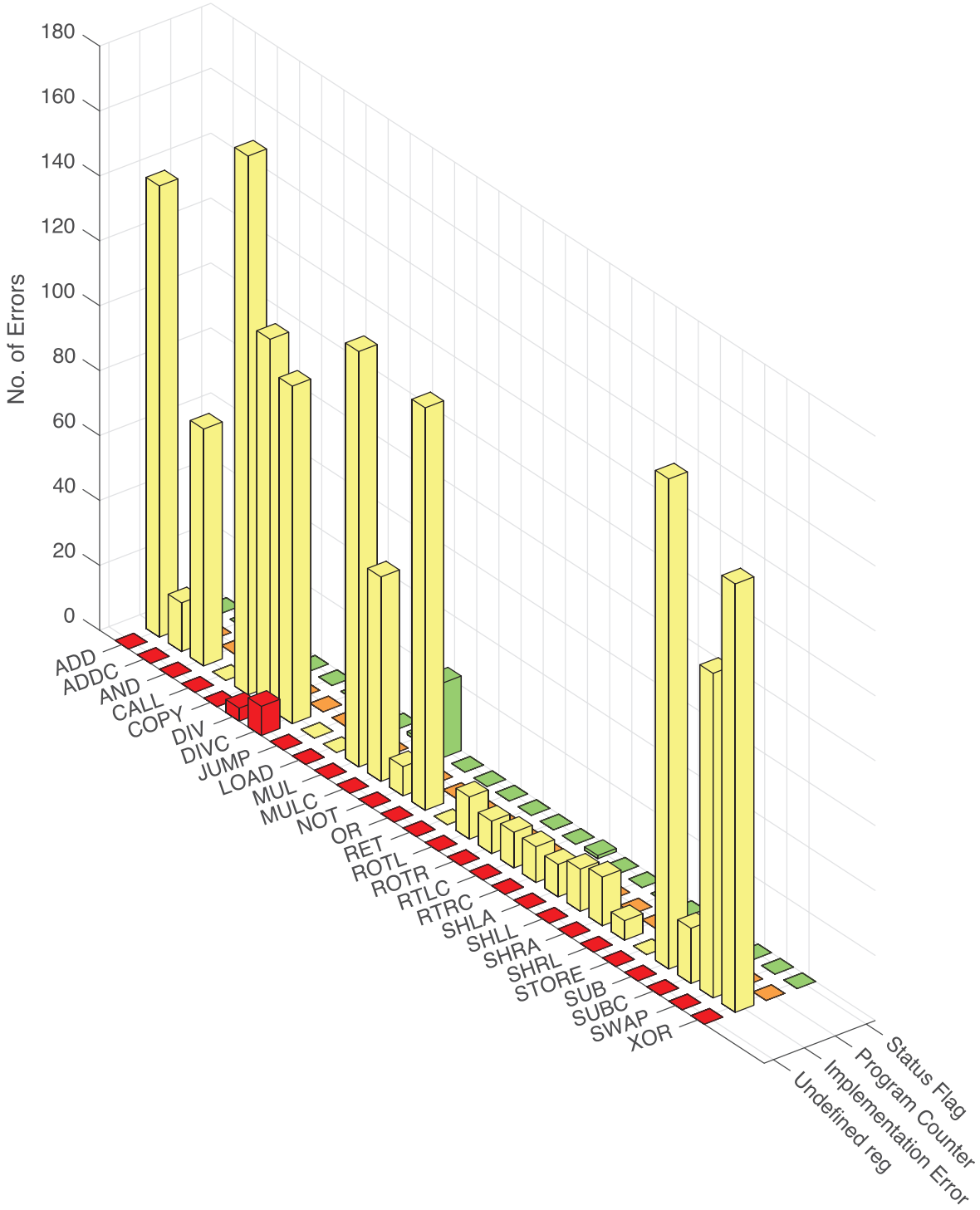

Figure V.8: Errors found in processor *axt* while executing test *T2* (mode M)

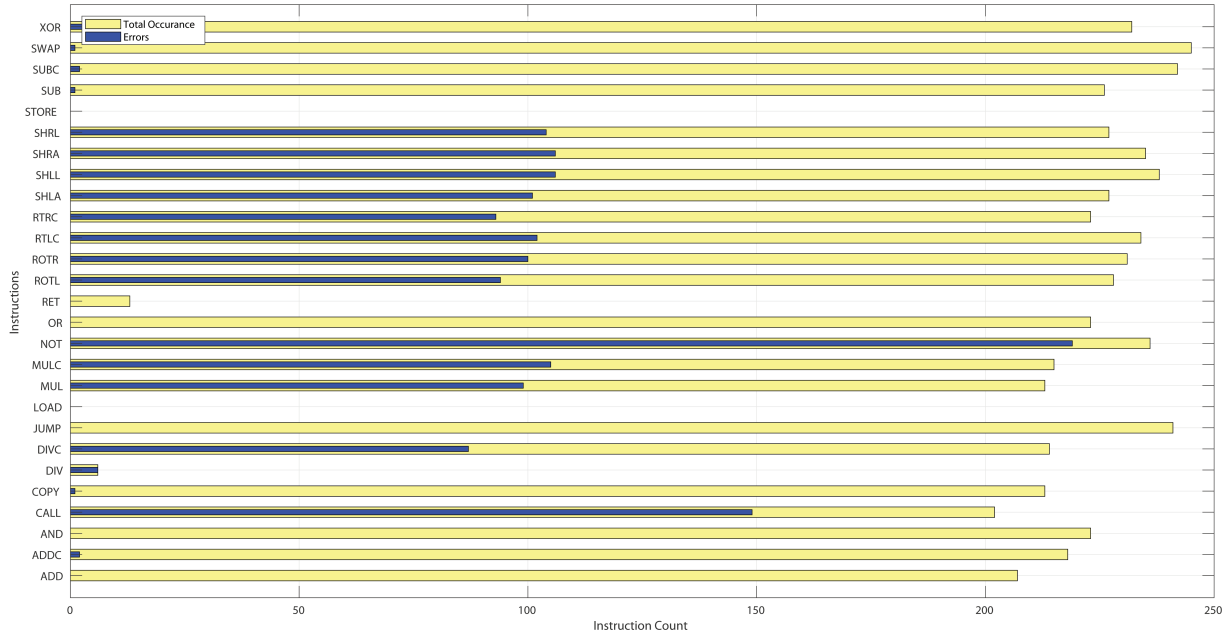

Figure V.9: Total Error count for test *T1* (mode MB) in processor *axt*

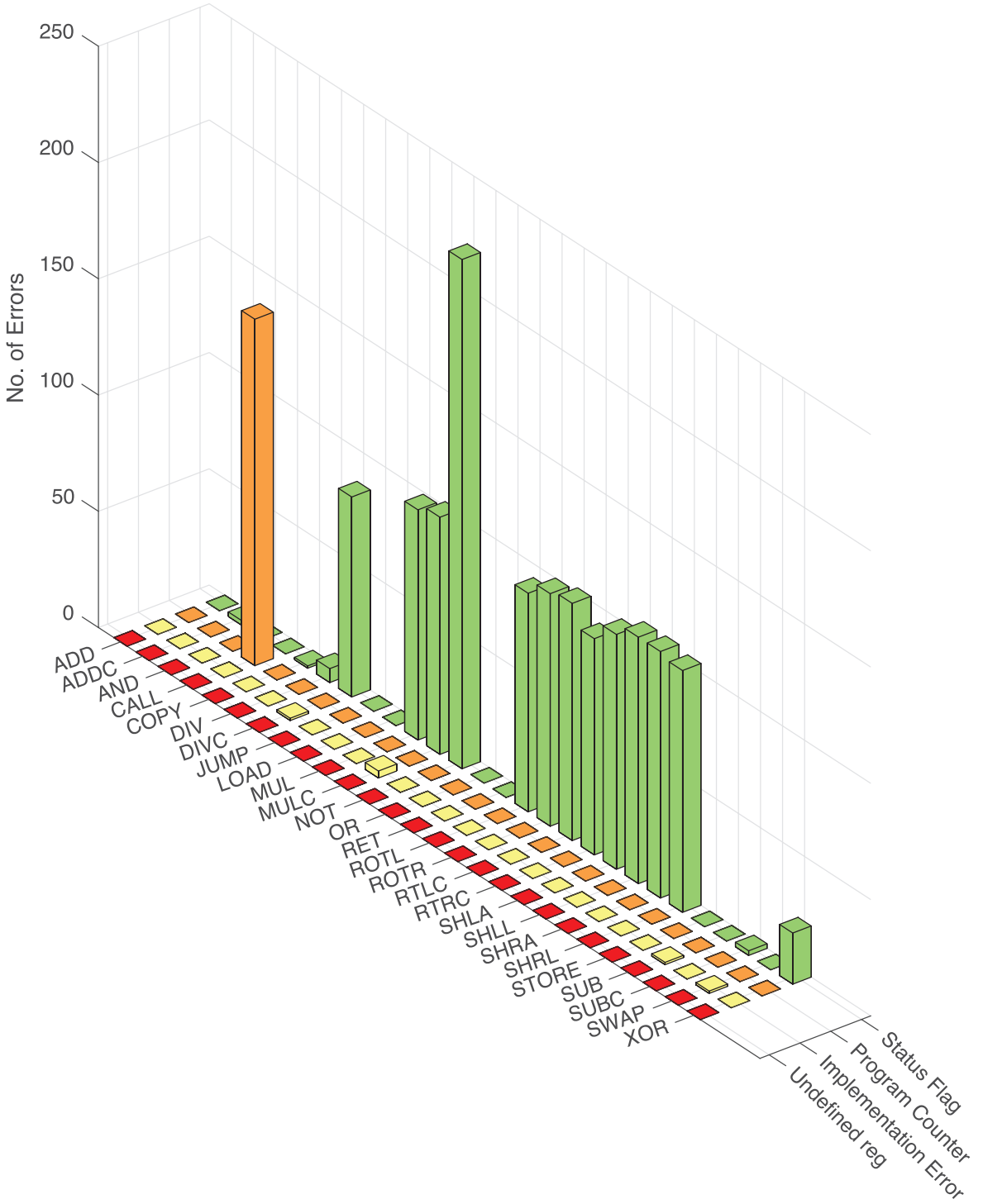

Figure V.10: Errors found in processor *axt* while executing test *T1* (mode MB)

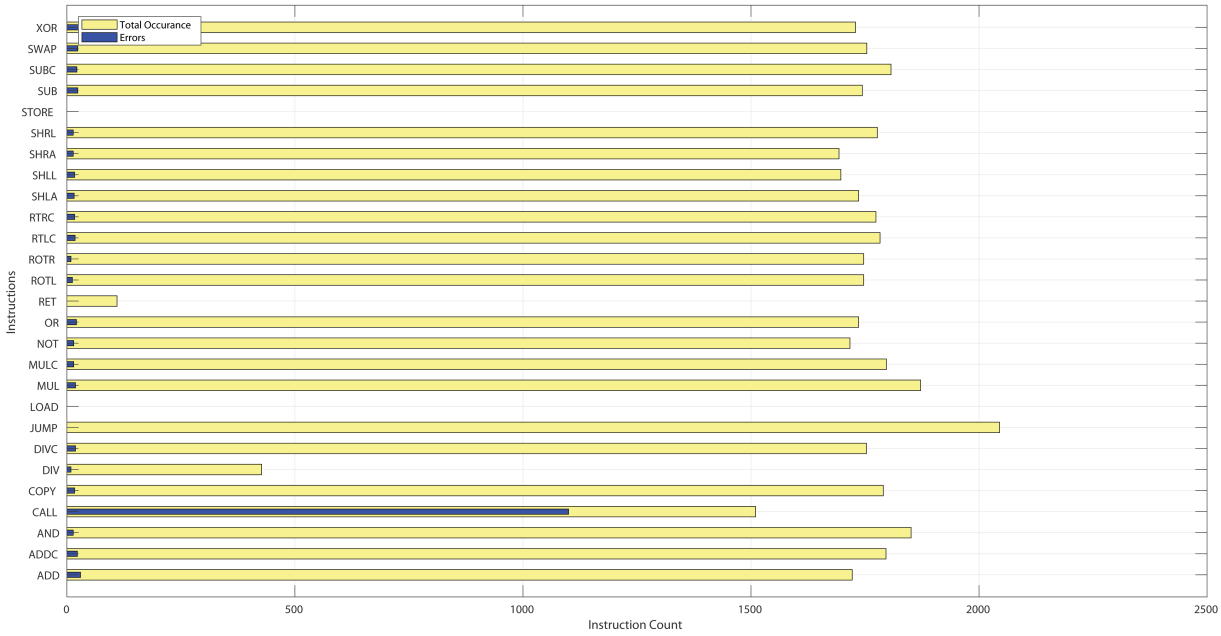

Figure V.11: Total Error count for test *T2* (mode MB) in processor *axt*

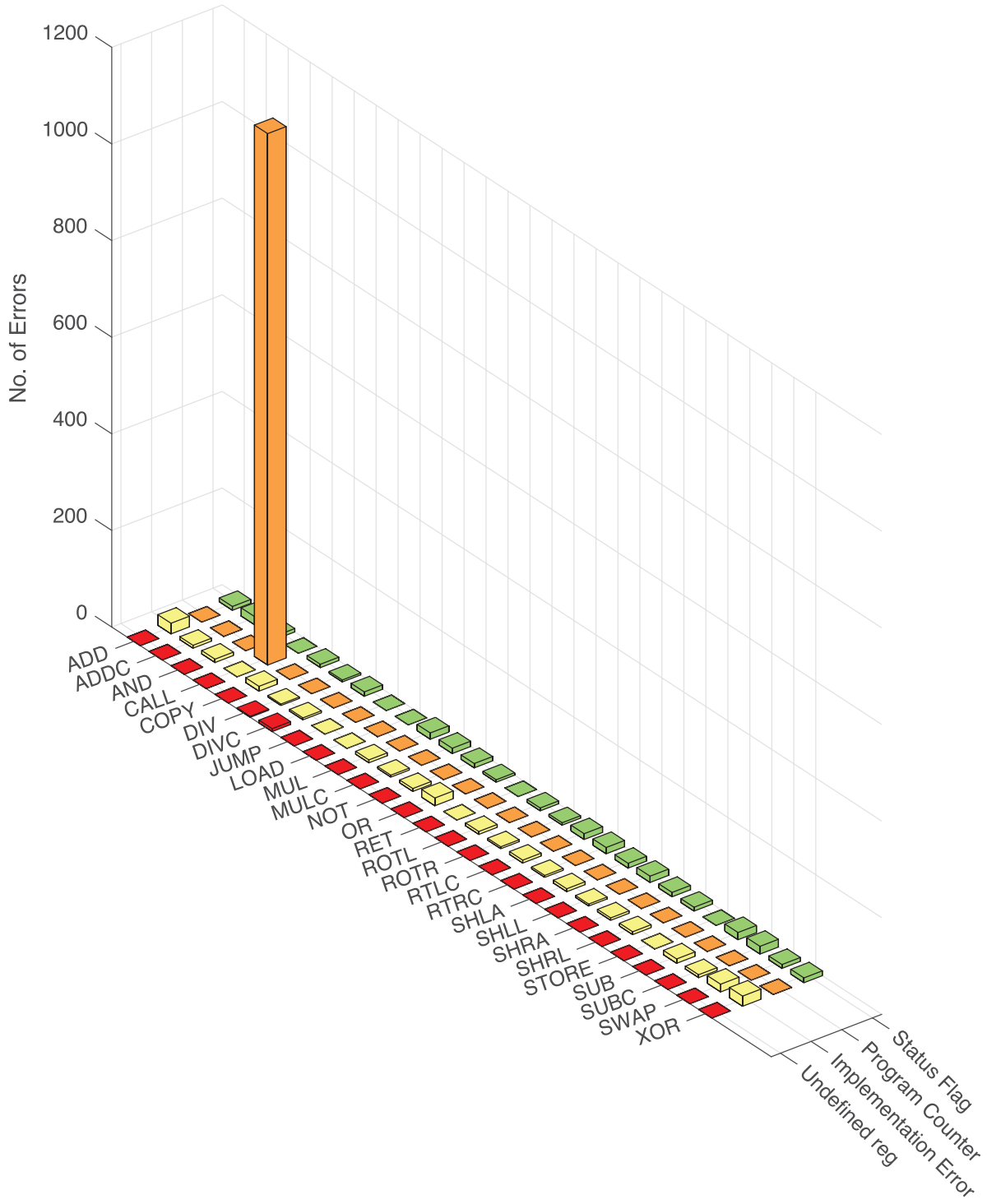

Figure V.12: Errors found in processor *axt* while executing test *T2* (mode MB)

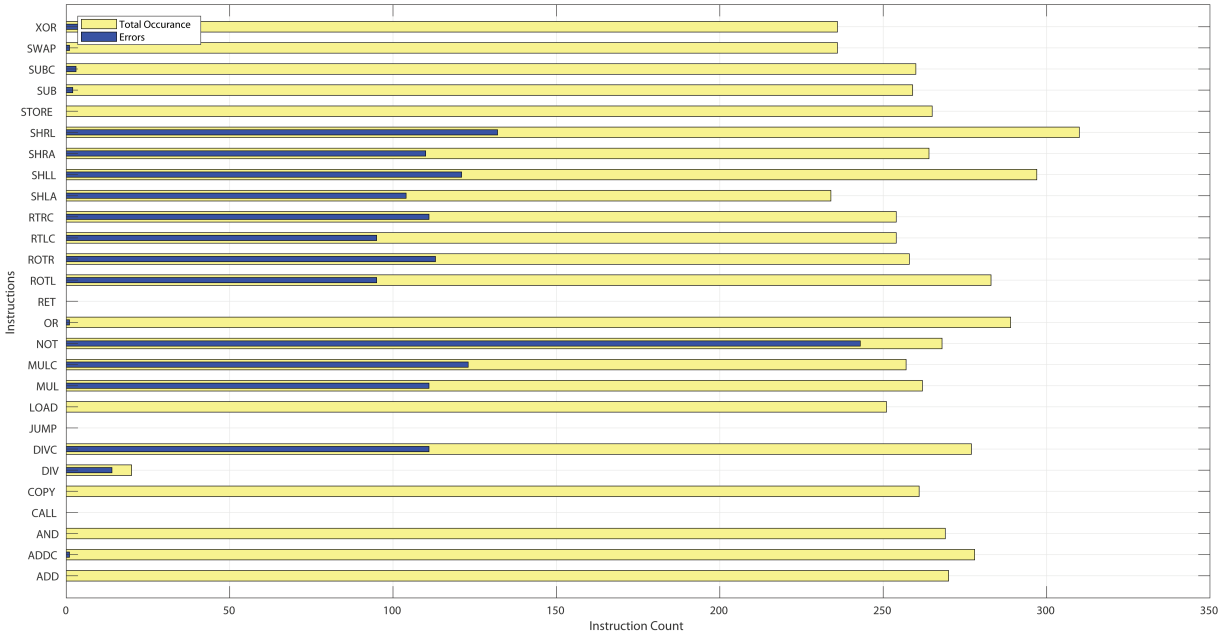

Figure V.13: Total Error count for test *T1* (mode MD) in processor *axt*

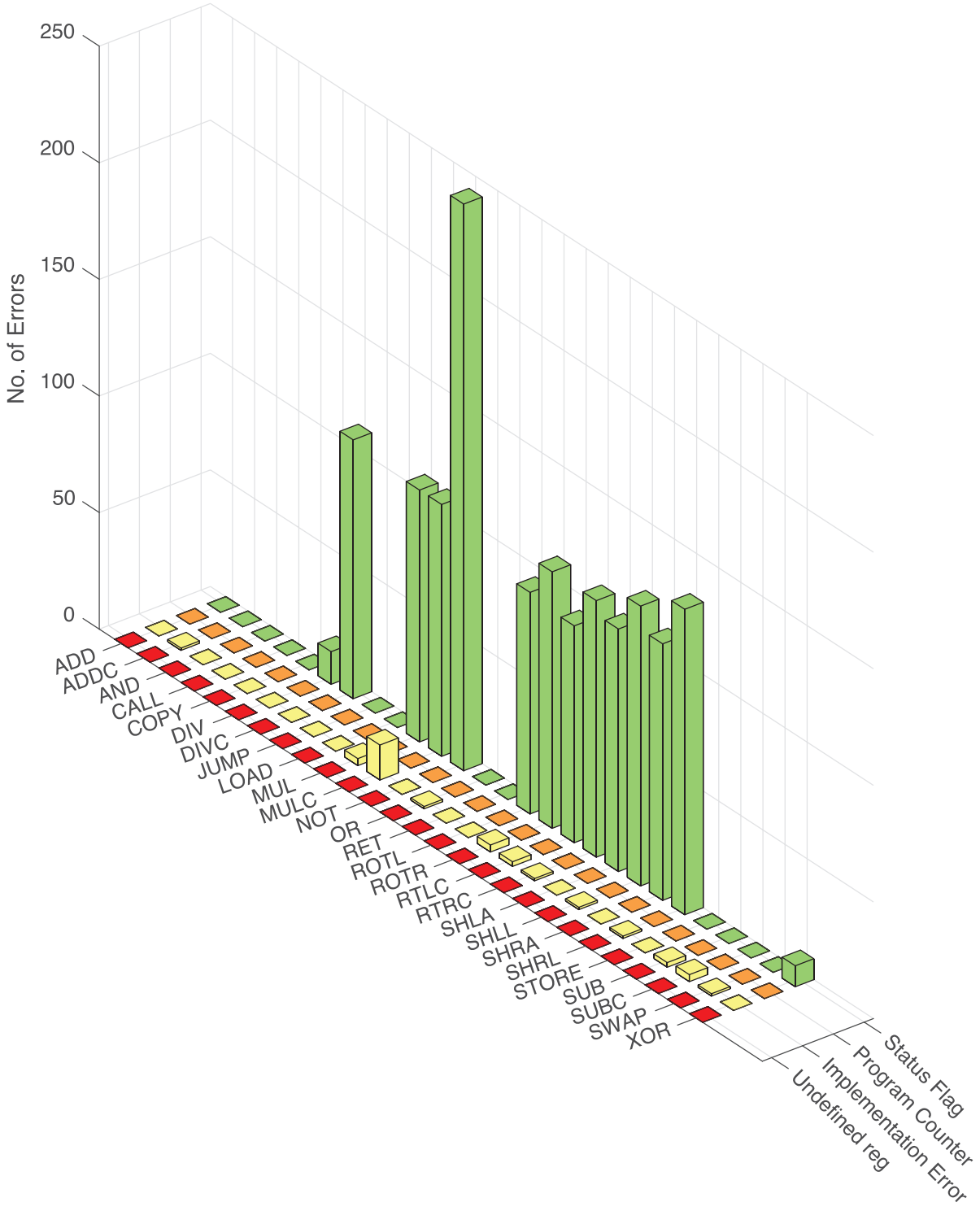

Figure V.14: Errors found in processor *axt* while executing test *T1* (mode MD)

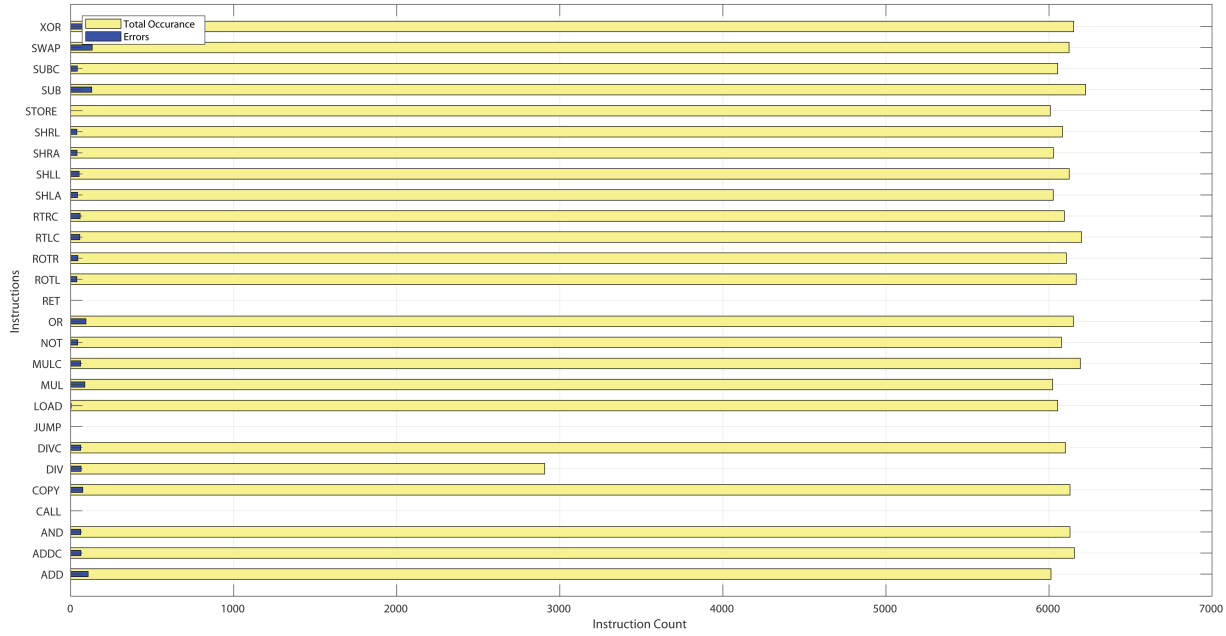

Figure V.15: Total Error count for test *T2* (mode MD) in processor *axt*

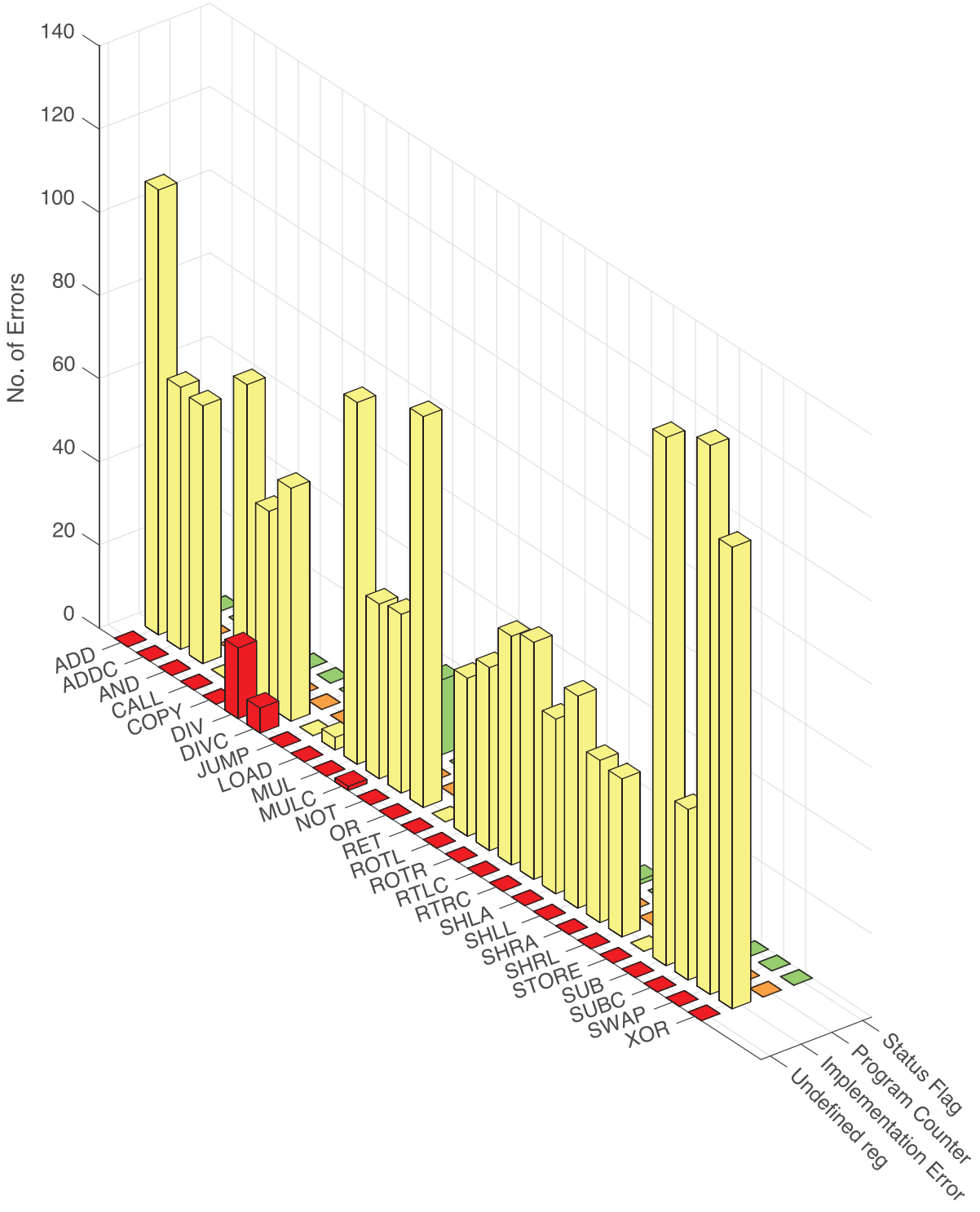

Figure V.16: Errors found in processor *axt* while executing test *T2* (mode MD)

V.2 Processor *dnm*

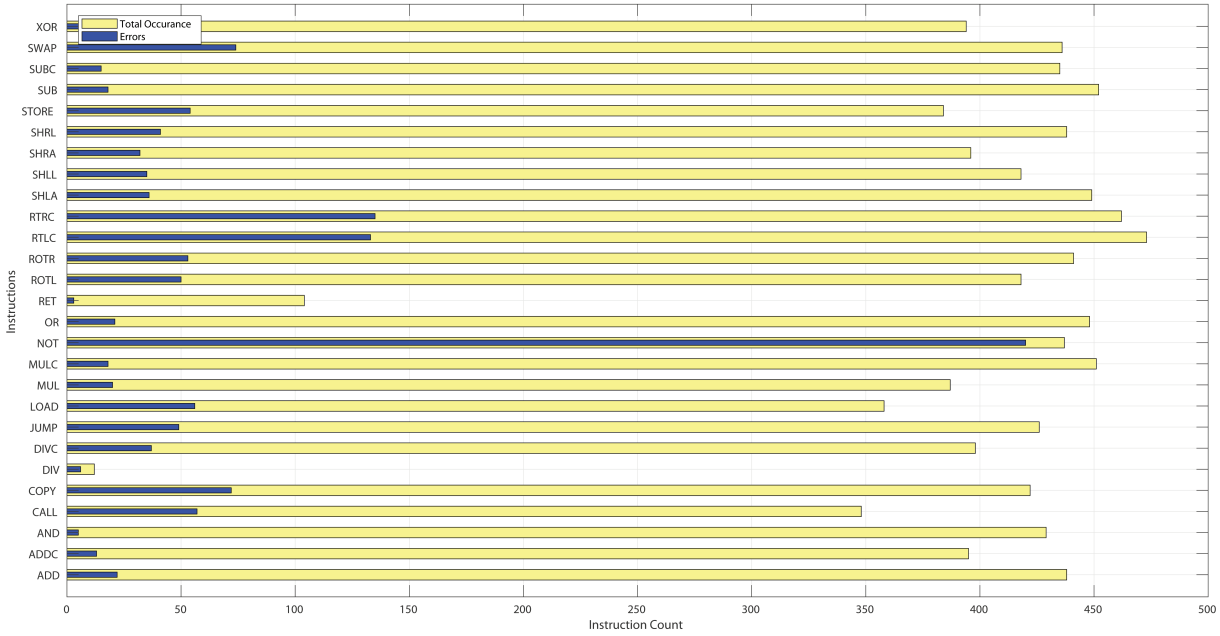

Figure V.17: Total Error count for test *T1* (mode A) in processor *dnm*

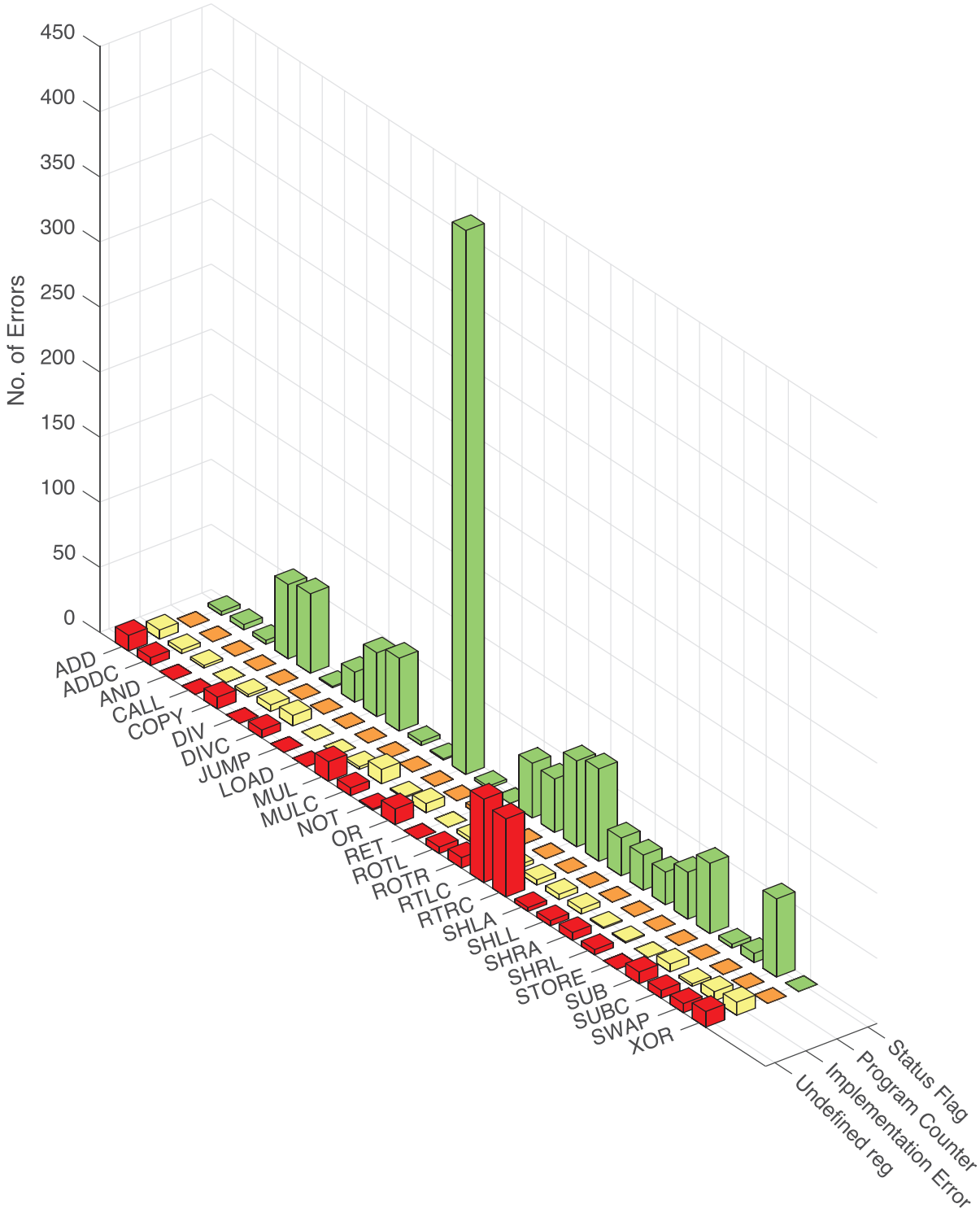

Figure V.18: Errors found in processor *dnm* while executing test *T1* (mode A)

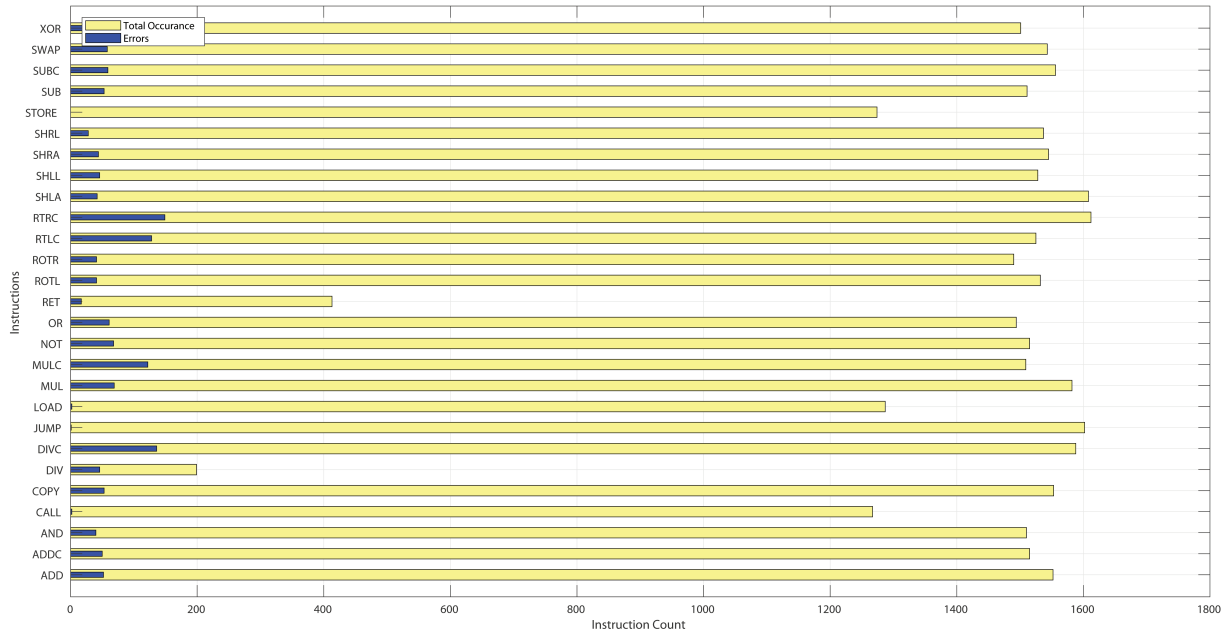

Figure V.19: Total Error count for test *T2* (mode A) in processor *dnm*

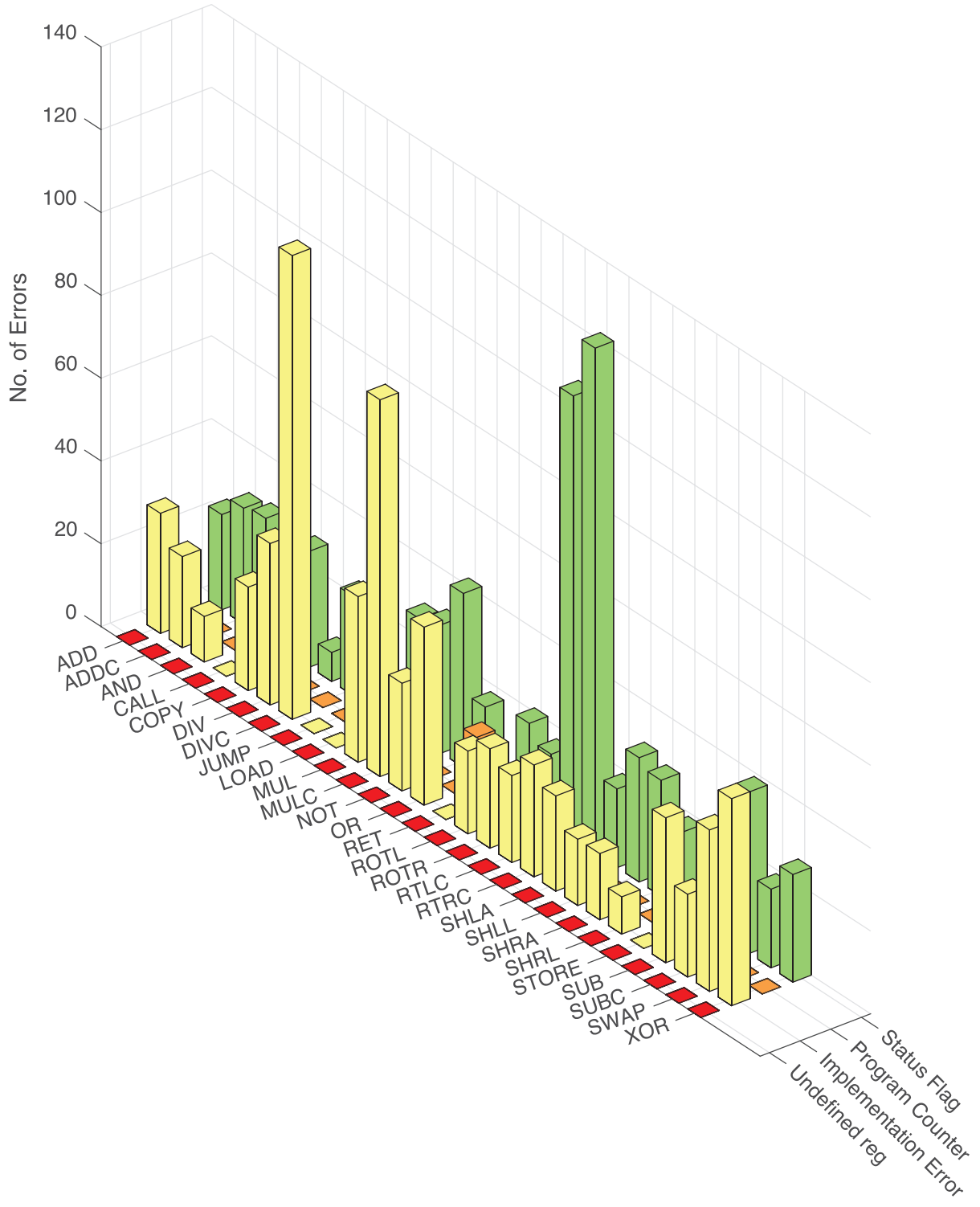

Figure V.20: Errors found in processor *dnm* while executing test *T2* (mode A)

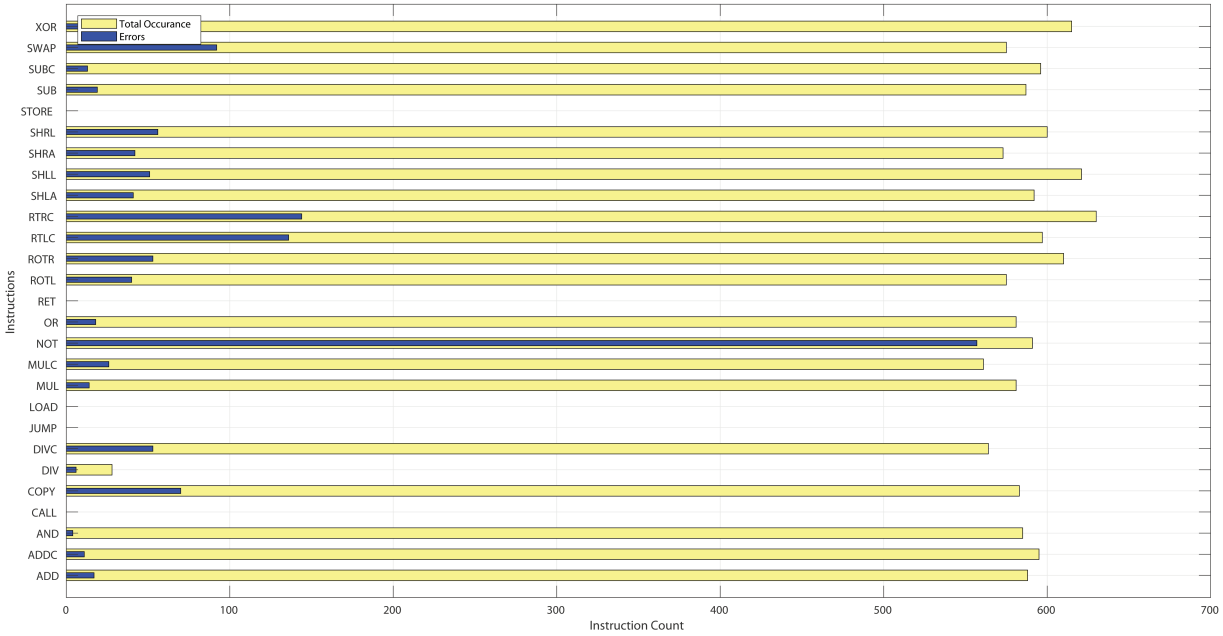

Figure V.21: Total Error count for test *T1* (mode M) in processor *dnm*

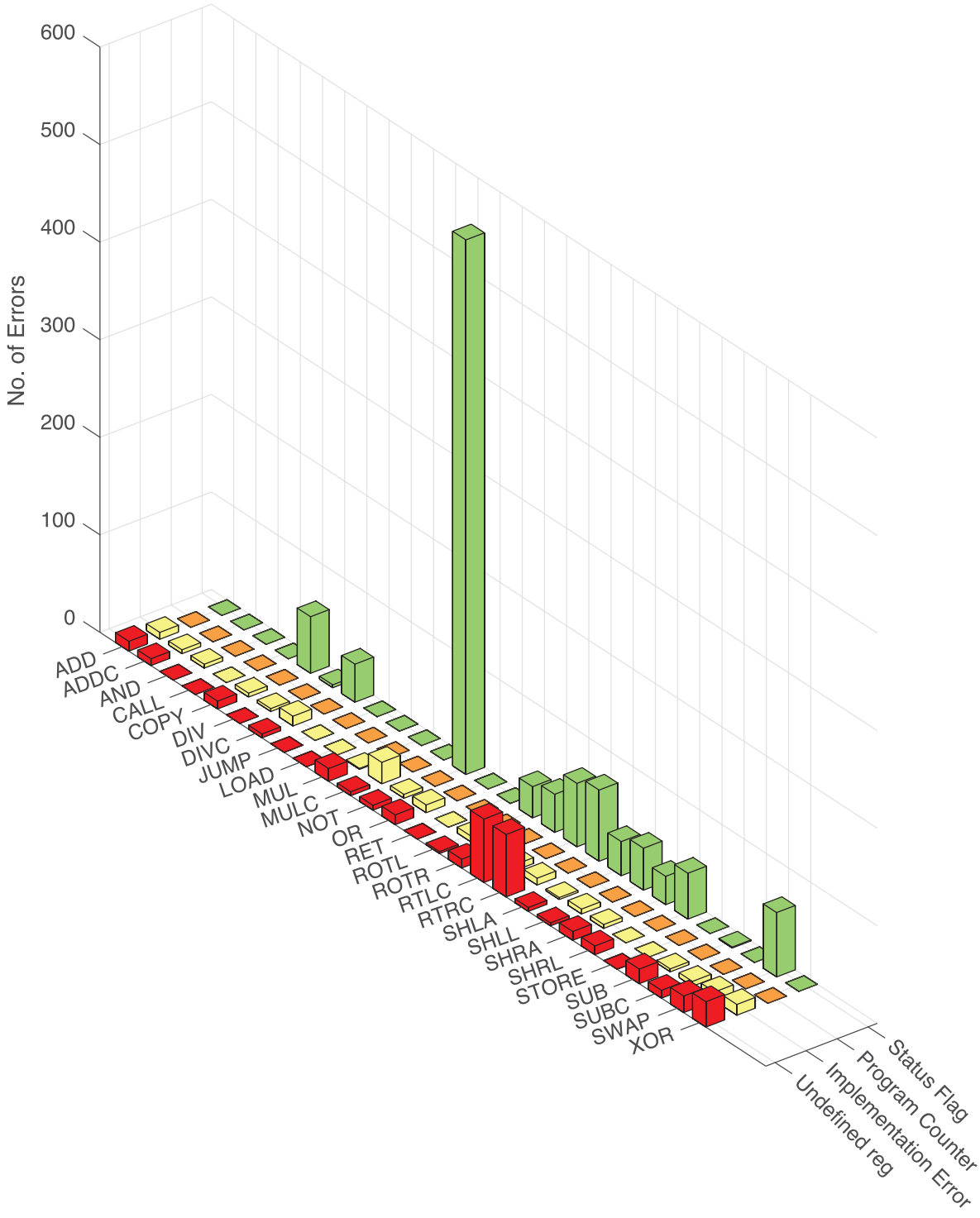

Figure V.22: Errors found in processor *dnm* while executing test *T1* (mode M)

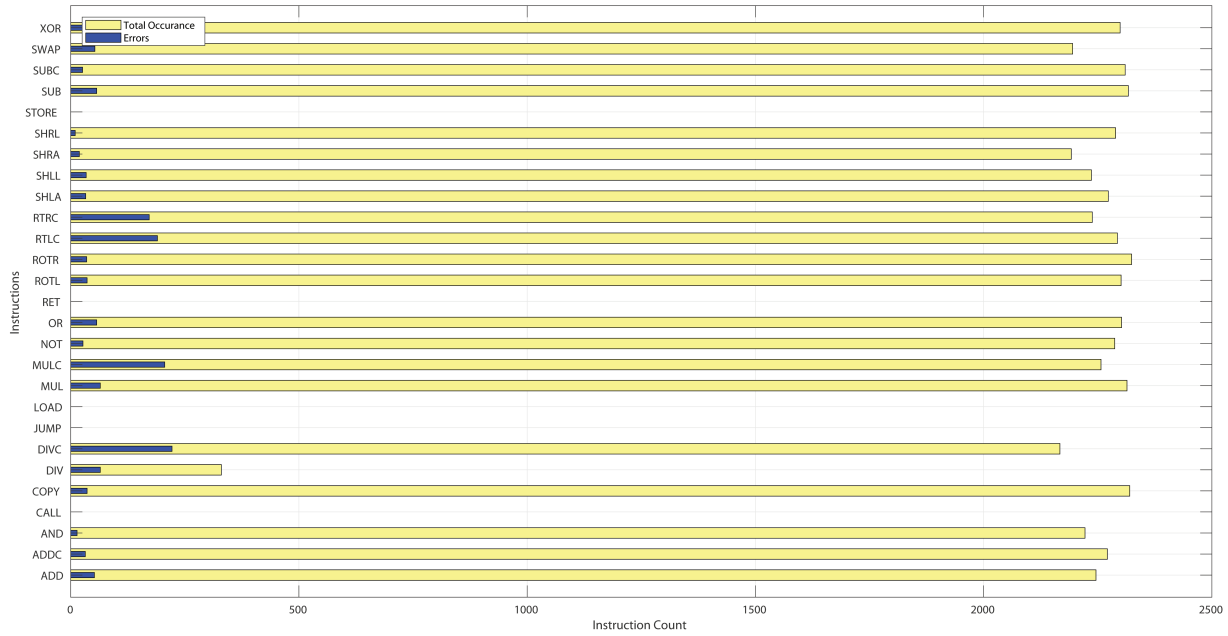

Figure V.23: Total Error count for test *T2* (mode M) in processor *dnm*

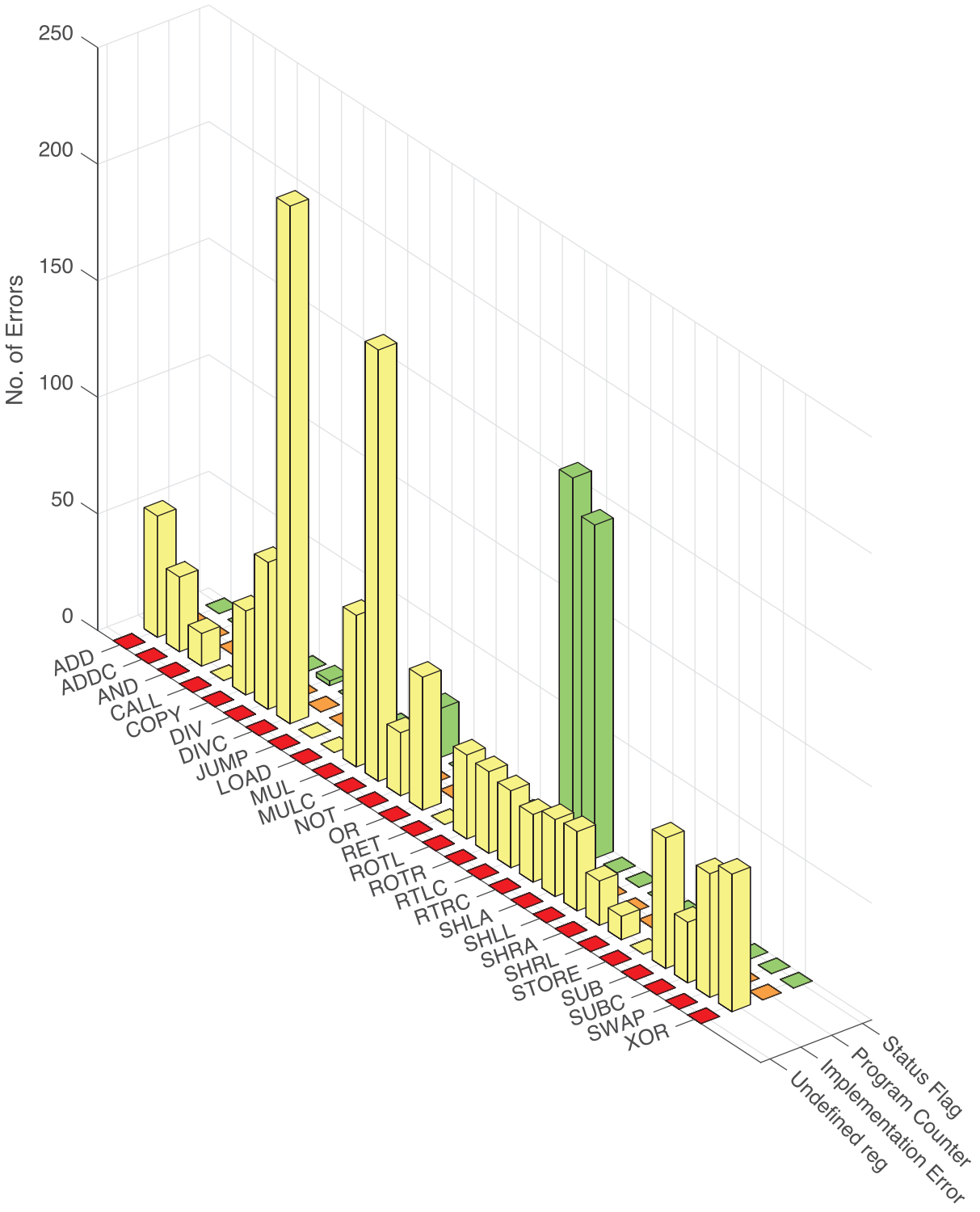

Figure V.24: Errors found in processor *dnm* while executing test *T2* (mode M)

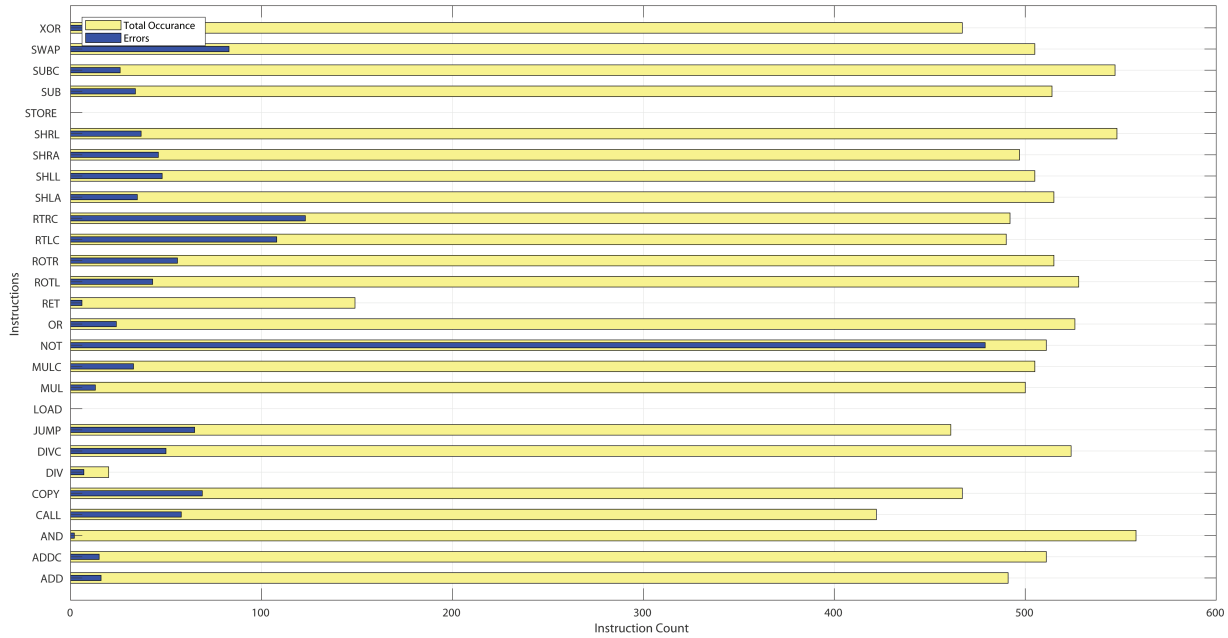

Figure V.25: Total Error count for test *T1* (mode MB) in processor *dnm*

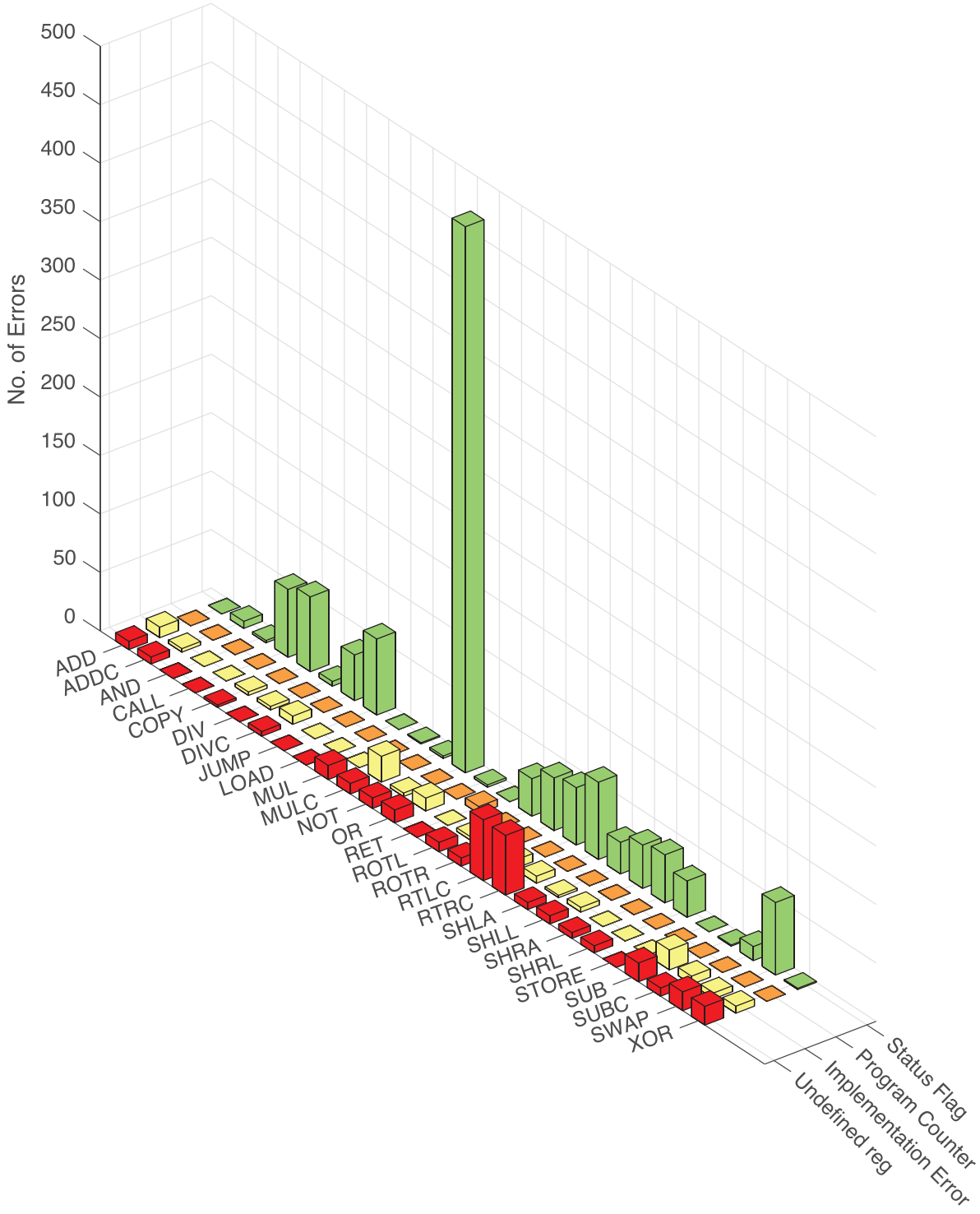

Figure V.26: Errors found in processor *dnm* while executing test *T1* (mode MB)

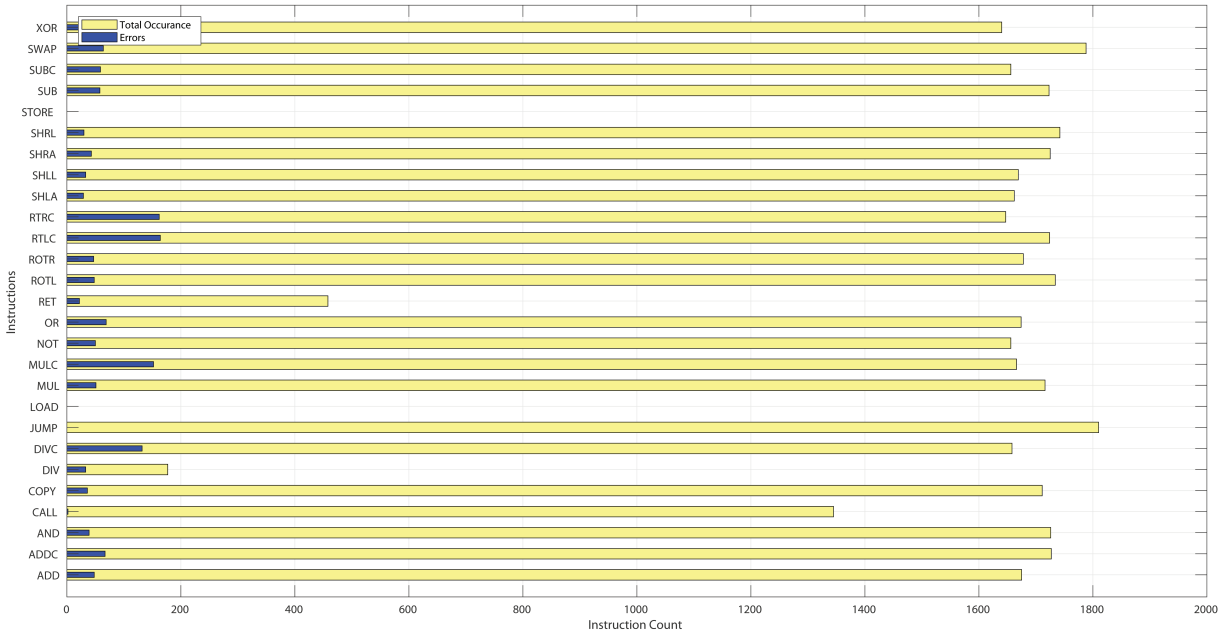

Figure V.27: Total Error count for test *T2* (mode MB) in processor *dnm*

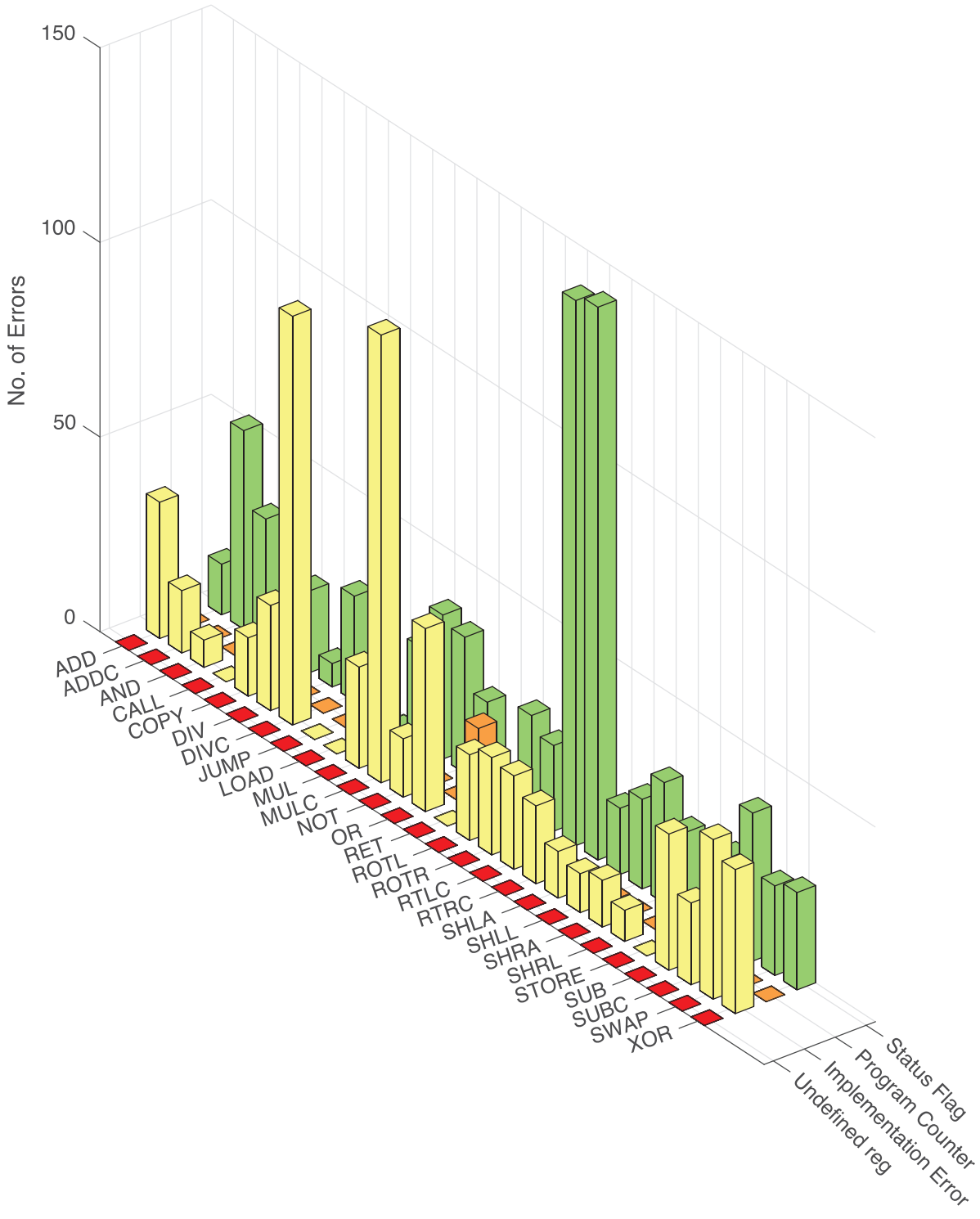

Figure V.28: Errors found in processor *dnm* while executing test *T2* (mode MB)

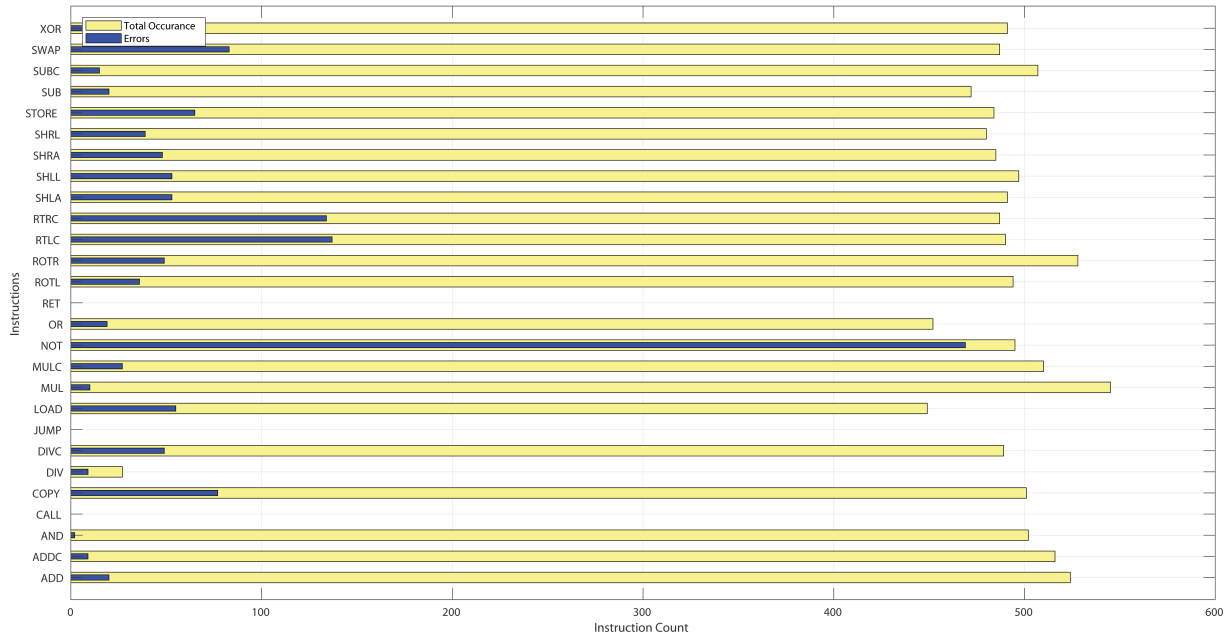

Figure V.29: Total Error count for test *T1* (mode MD) in processor *dnm*

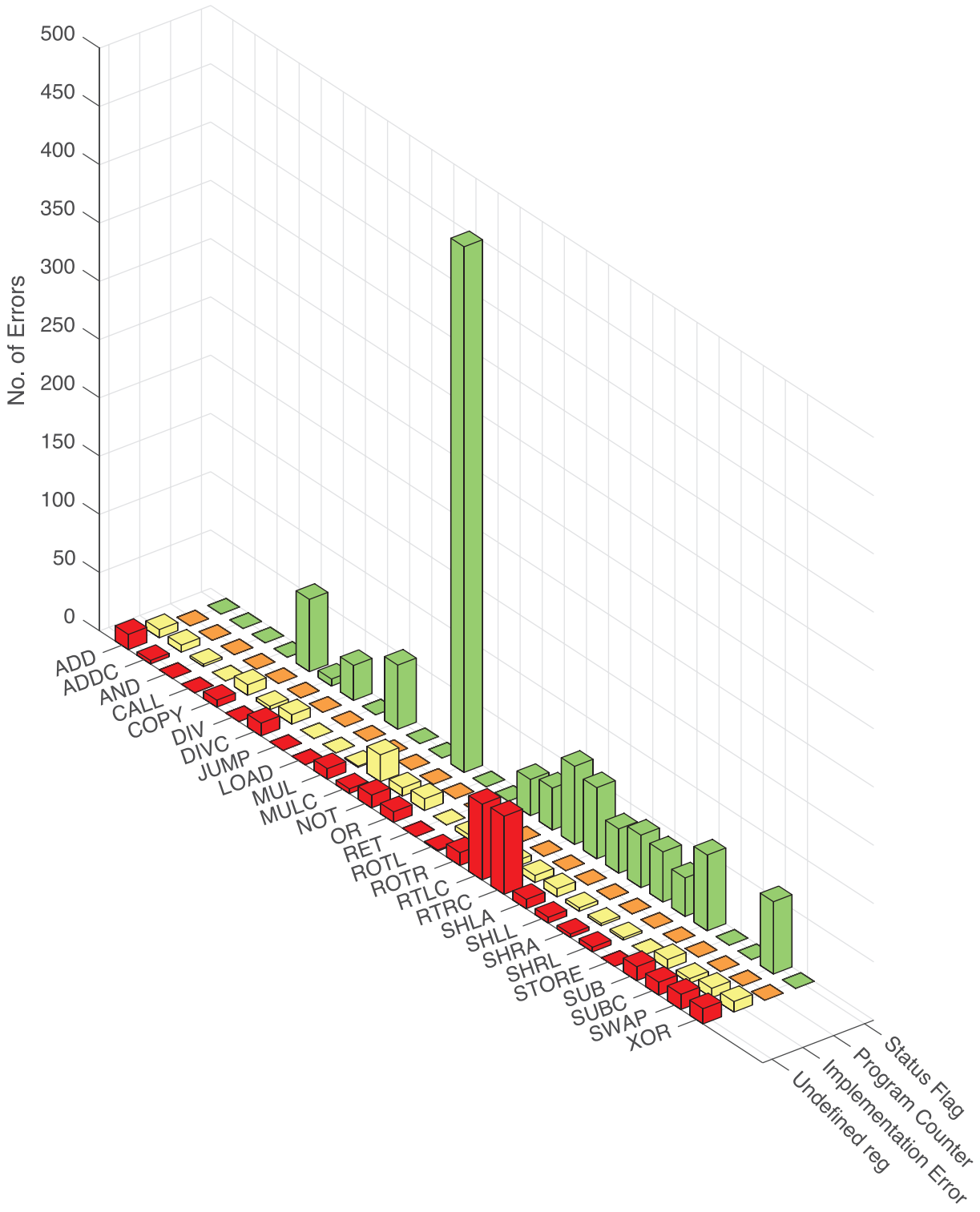

Figure V.30: Errors found in processor *dnm* while executing test *T1* (mode MD)

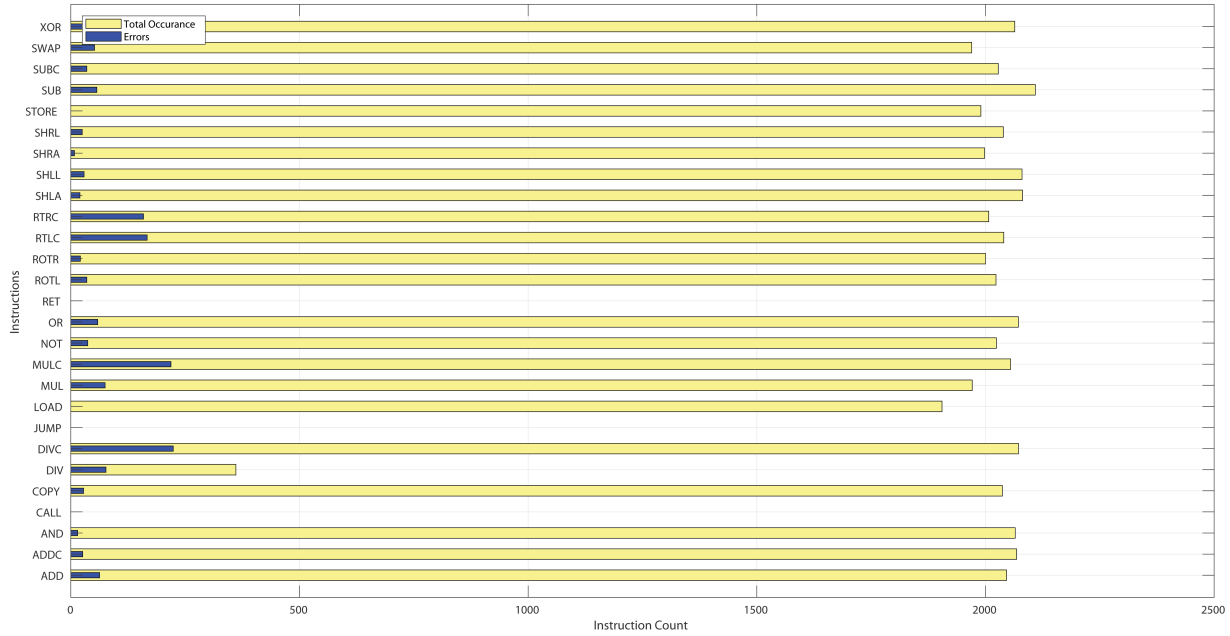

Figure V.31: Total Error count for test *T2* (mode MD) in processor *dnm*

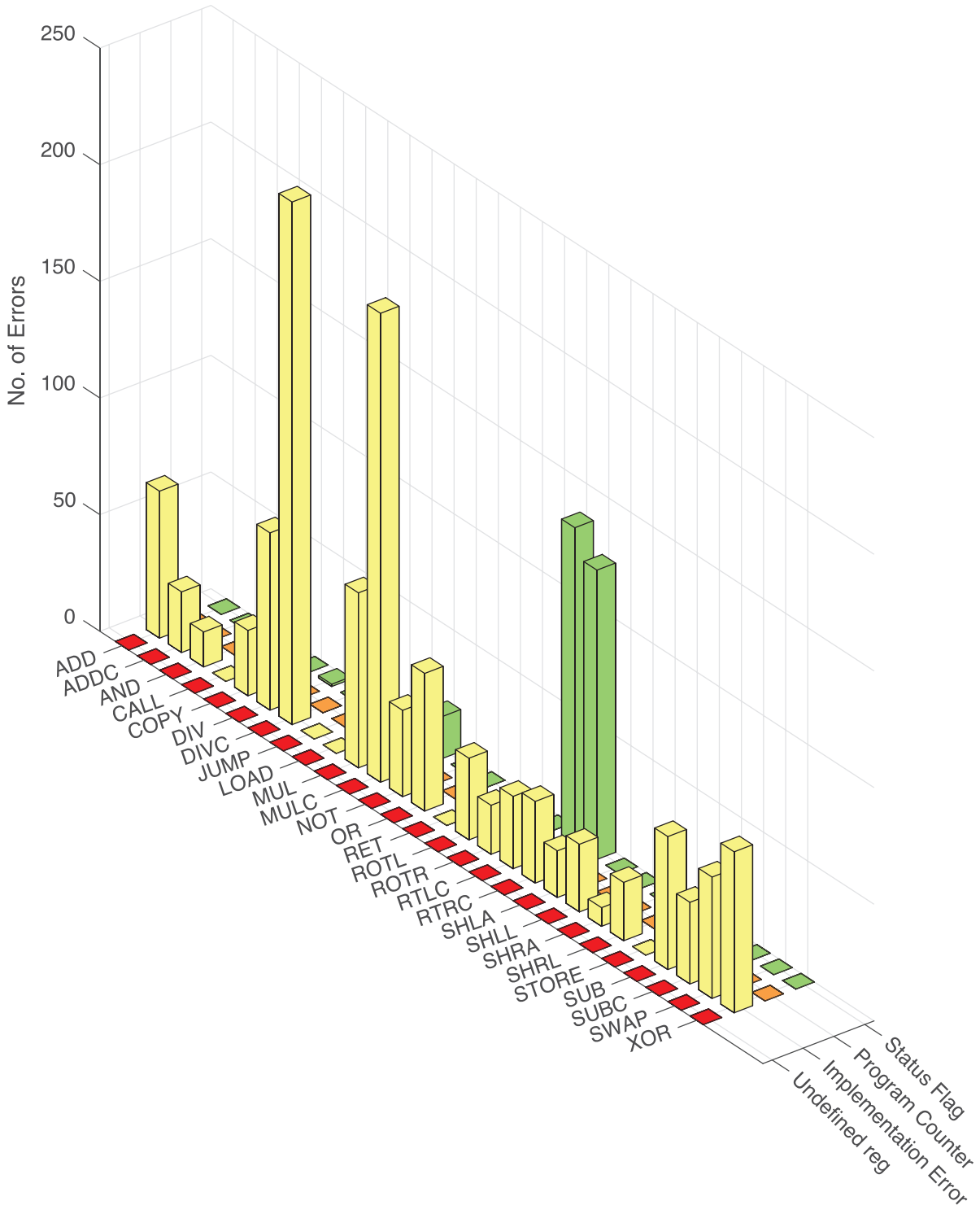

Figure V.32: Errors found in processor *dnm* while executing test *T2* (mode MD)

V.3 Processor *nxp*

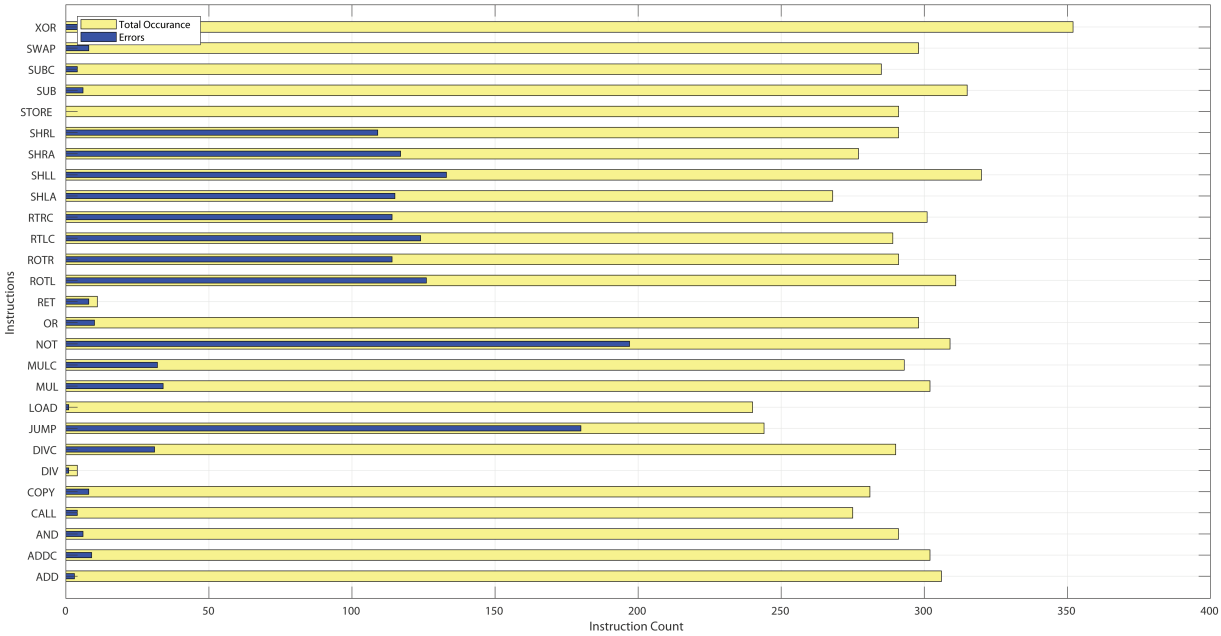

Figure V.33: Total Error count for test *T1* (mode A) in processor *nxp*

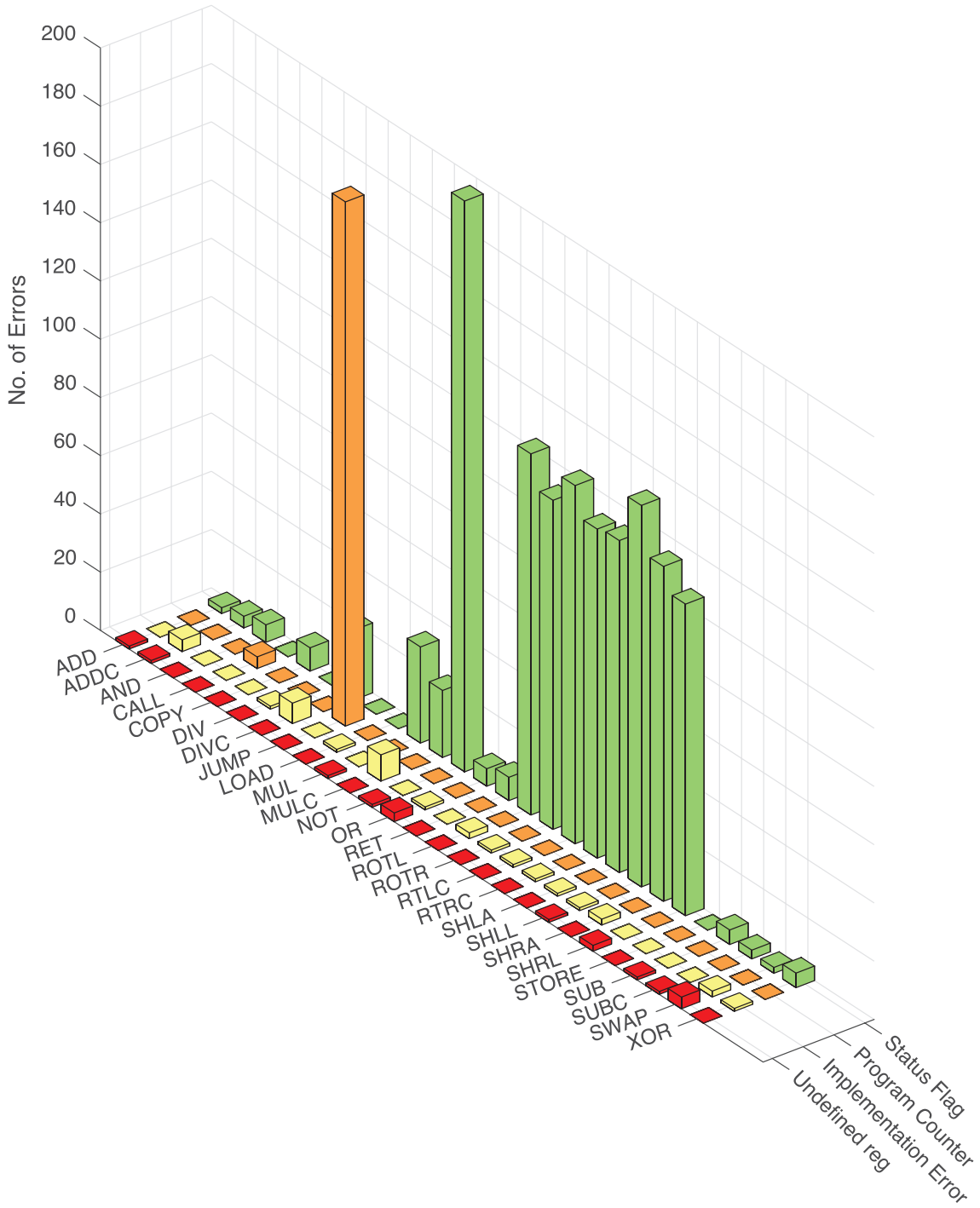

Figure V.34: Errors found in processor *nxp* while executing test *T1* (mode A)

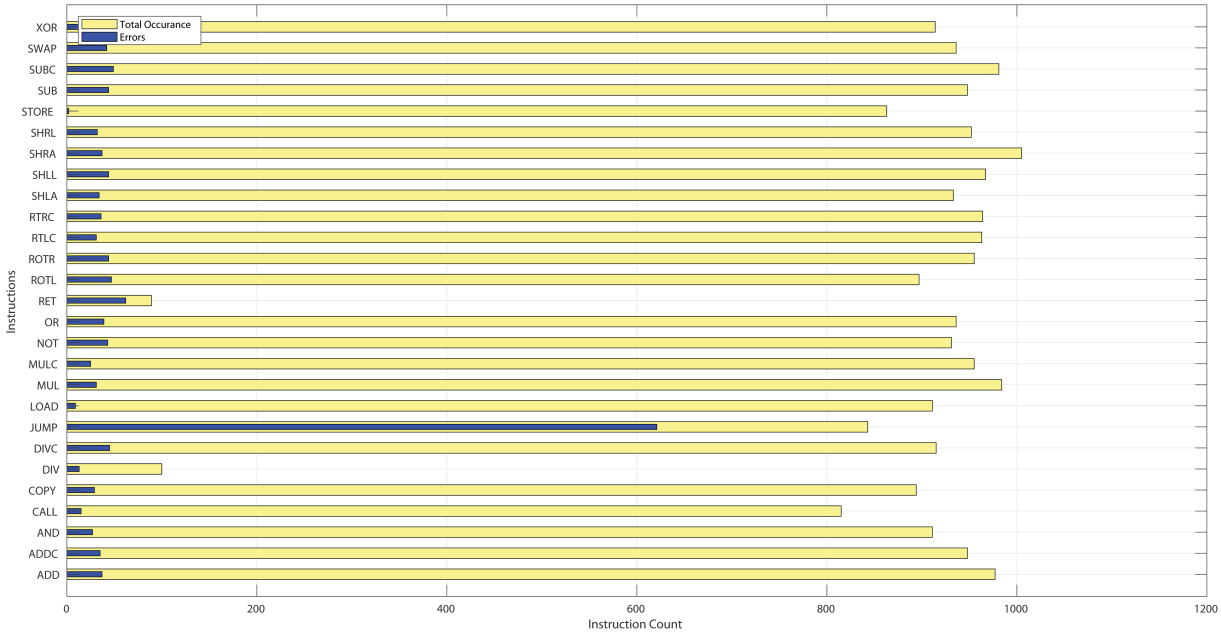

Figure V.35: Total Error count for test *T2* (mode A) in processor *nxp*

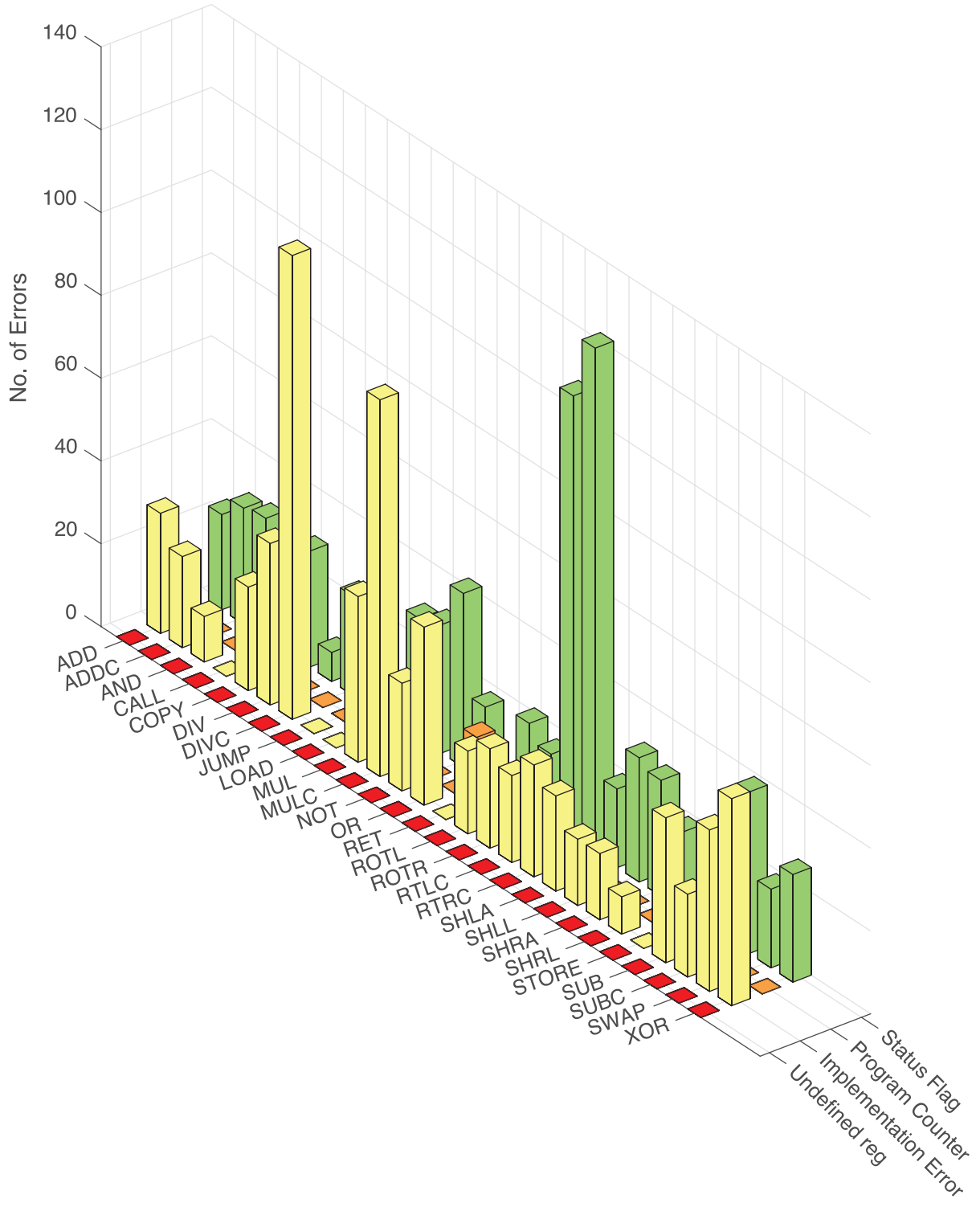

Figure V.36: Errors found in processor *nxp* while executing test *T2* (mode A)

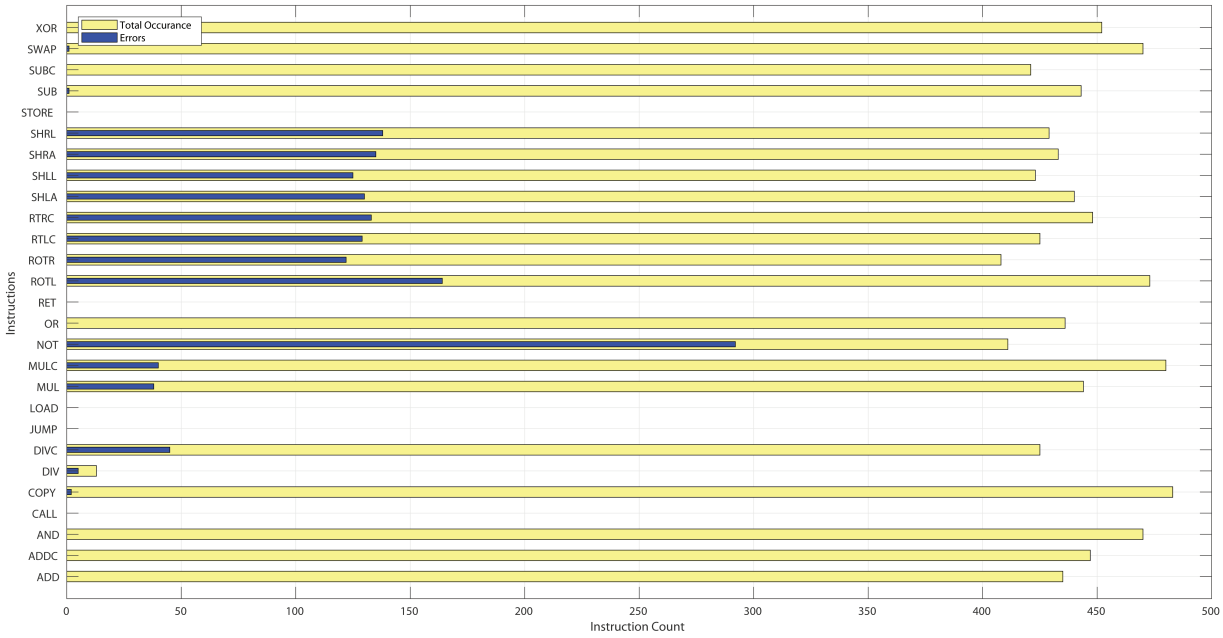

Figure V.37: Total Error count for test *T1* (mode M) in processor *nxp*

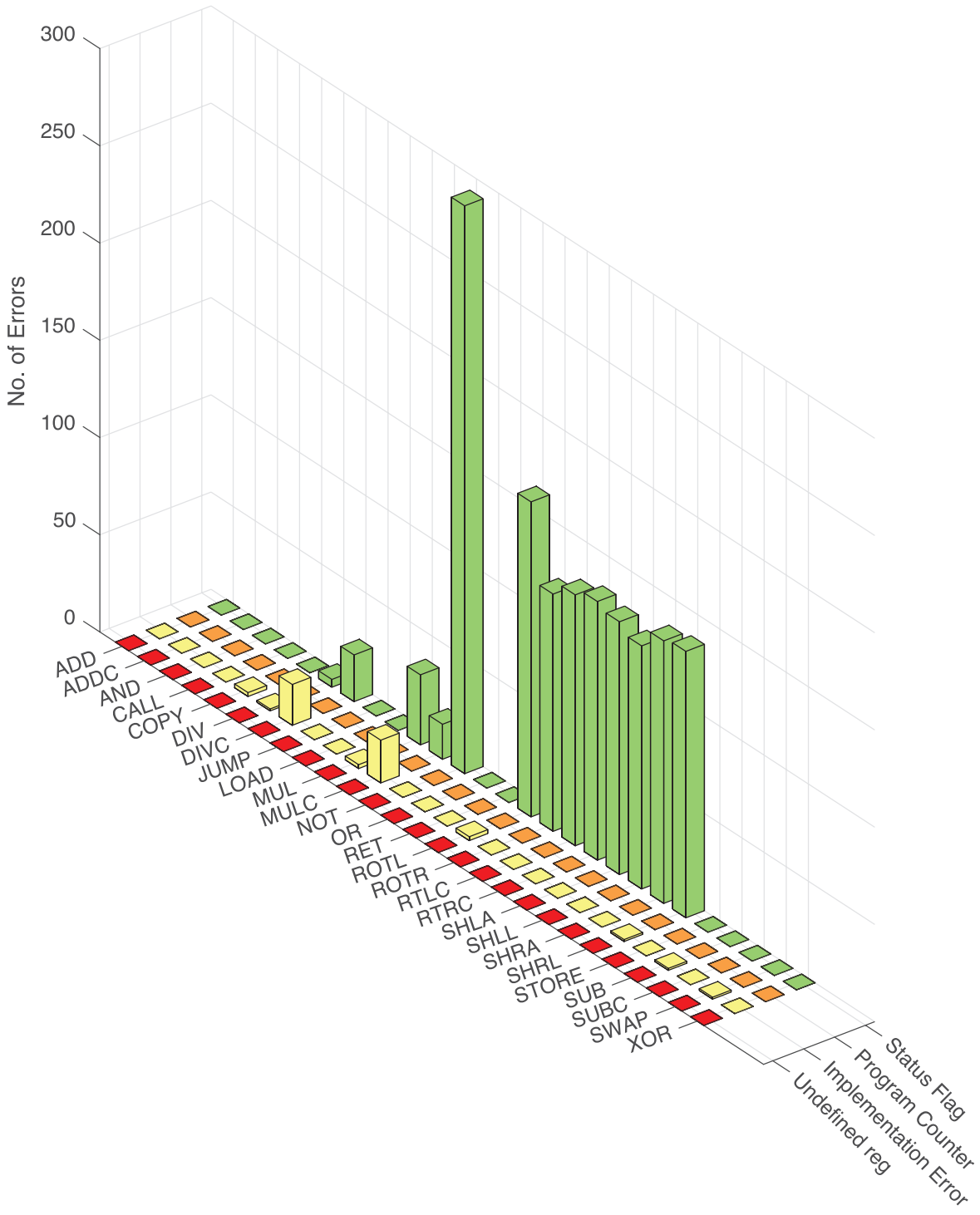

Figure V.38: Errors found in processor *nxp* while executing test *T1* (mode M)

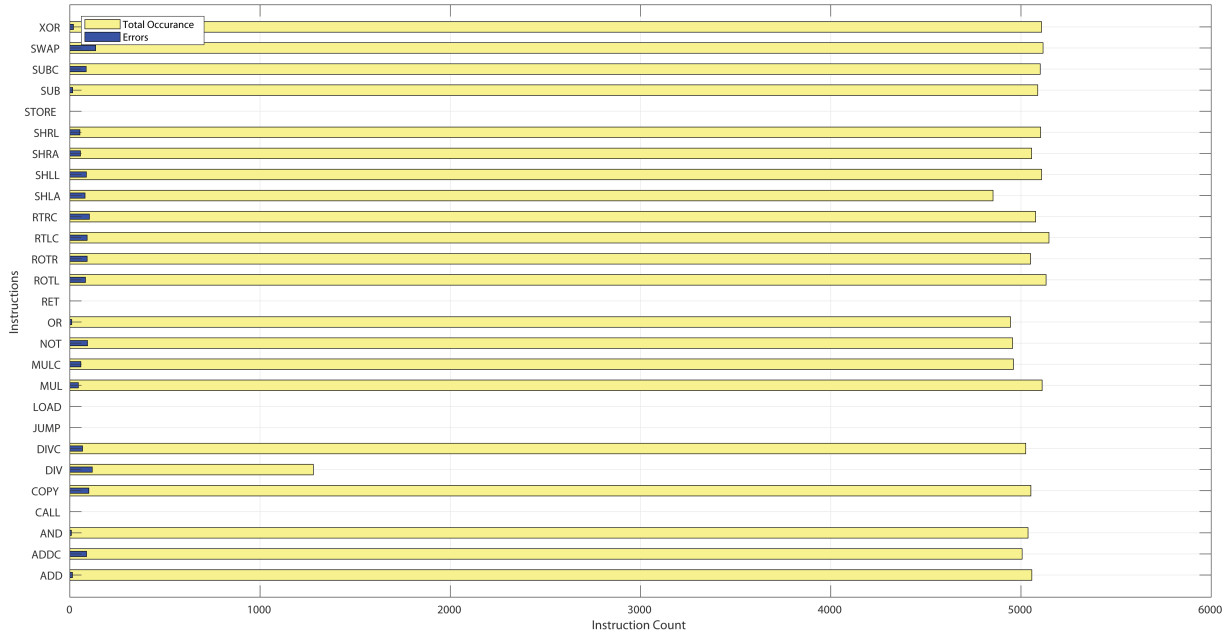

Figure V.39: Total Error count for test *T2* (mode M) in processor *nxp*

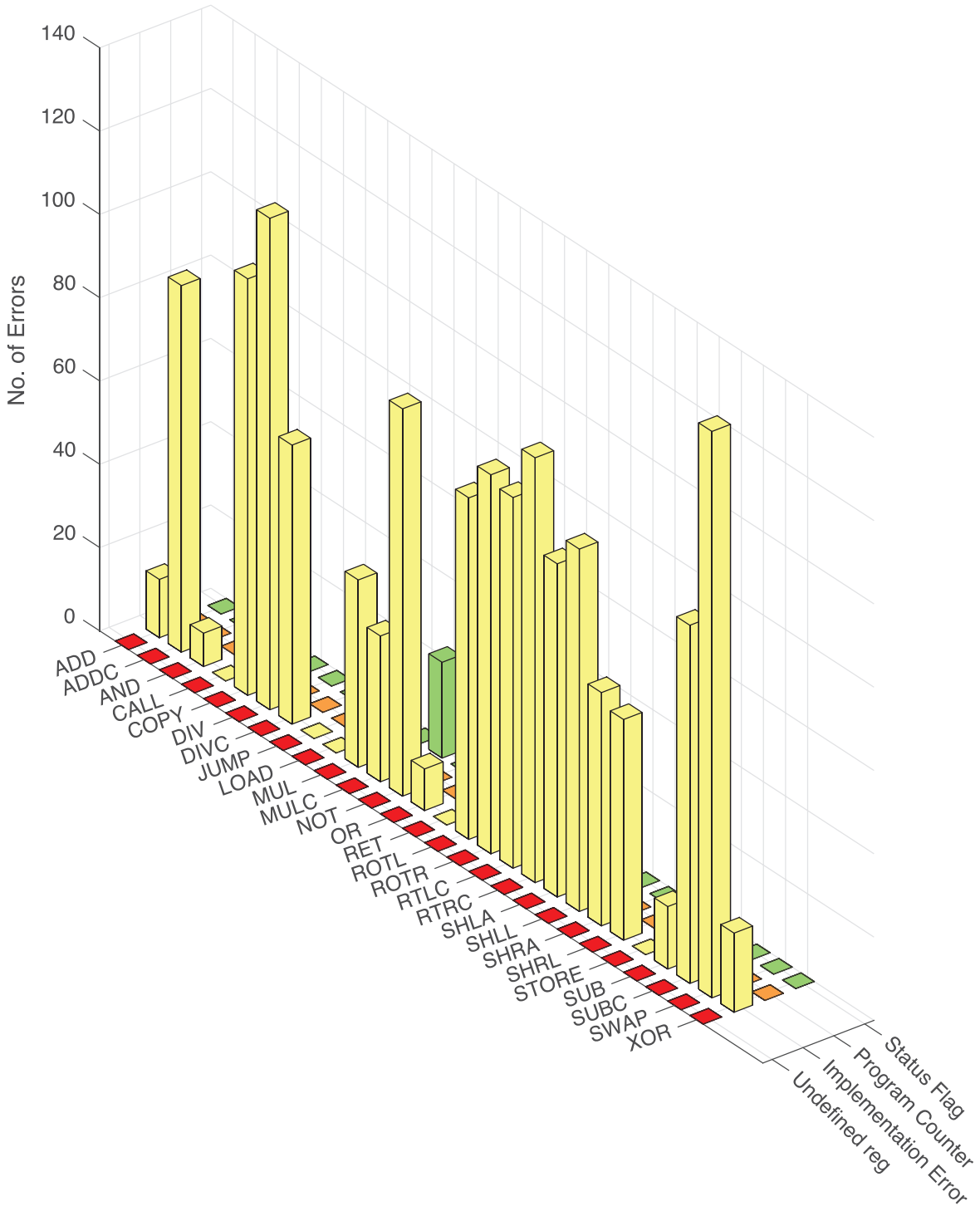

Figure V.40: Errors found in processor *nxp* while executing test *T2* (mode M)

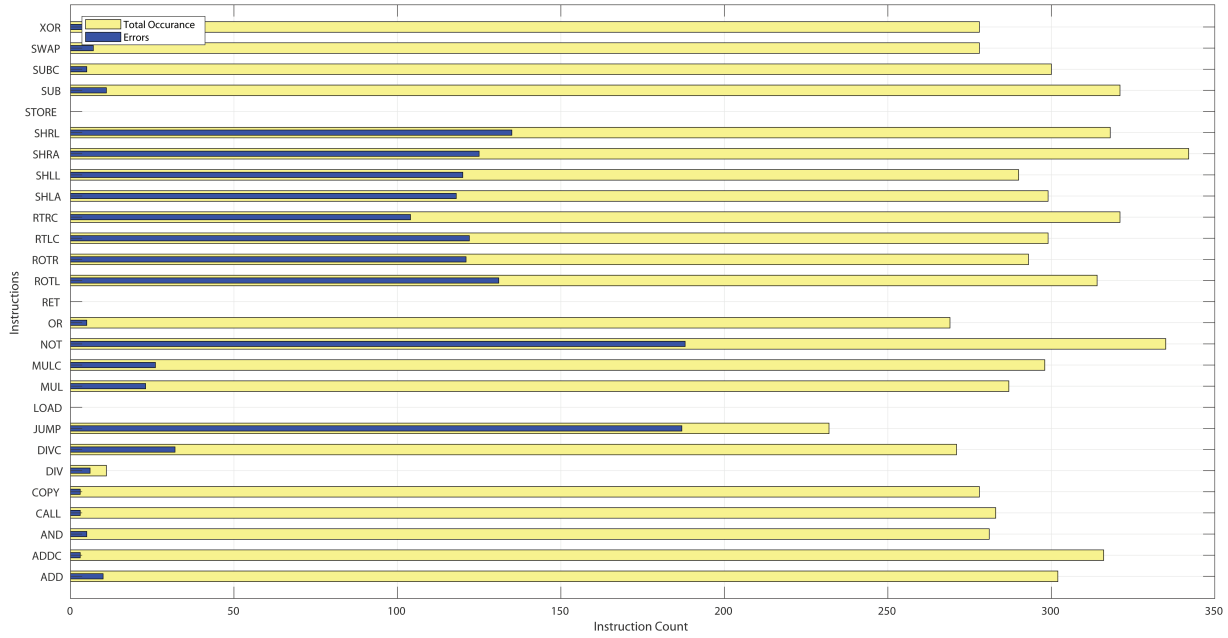

Figure V.41: Total Error count for test *T1* (mode MB) in processor *nxp*

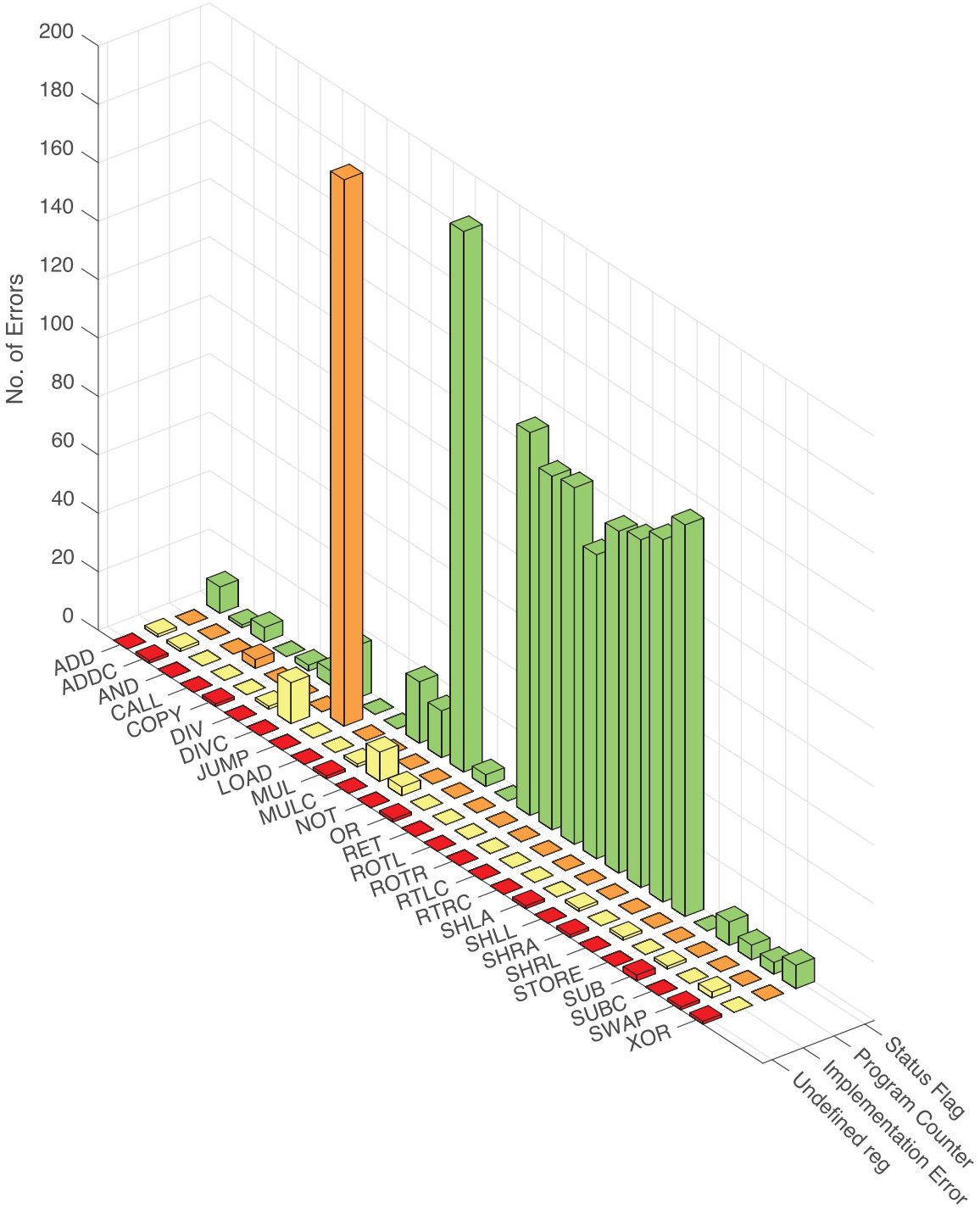

Figure V.42: Errors found in processor *nxp* while executing test *T1* (mode MB)

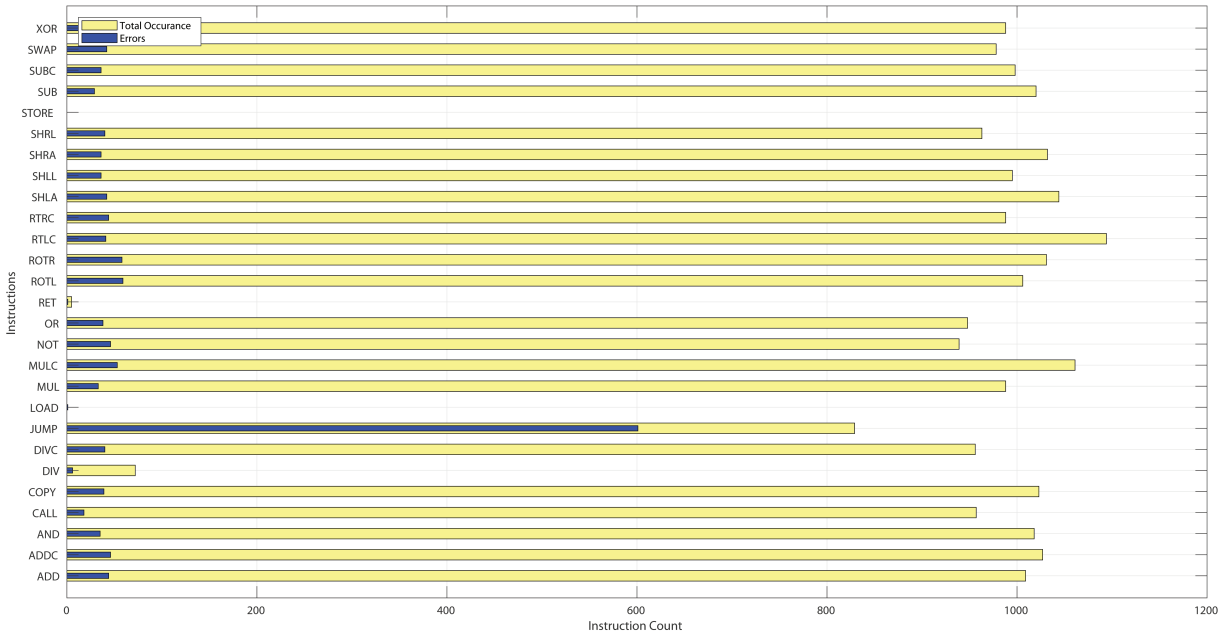

Figure V.43: Total Error count for test *T2* (mode MB) in processor *nxp*

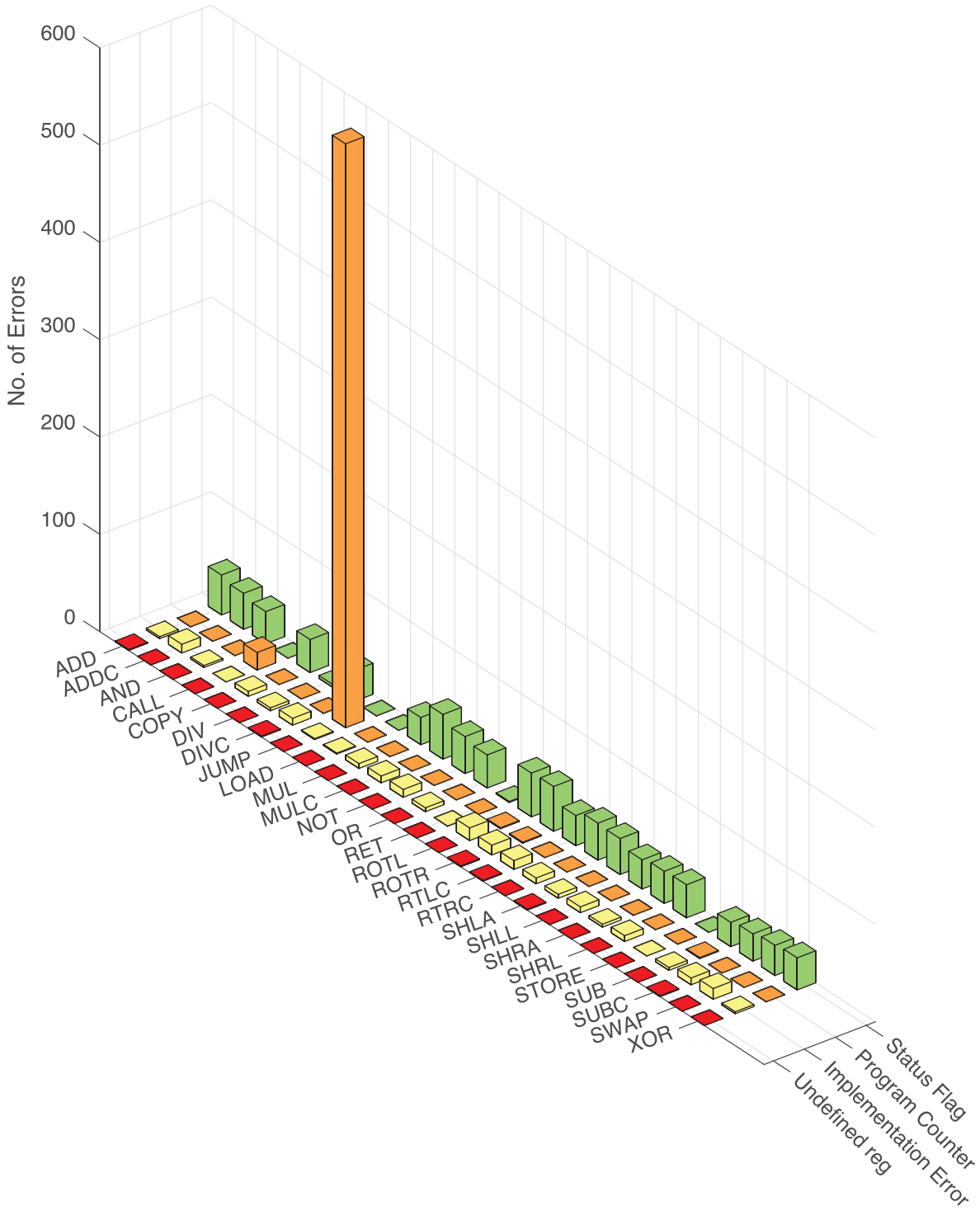

Figure V.44: Errors found in processor *nxp* while executing test *T2* (mode MB)

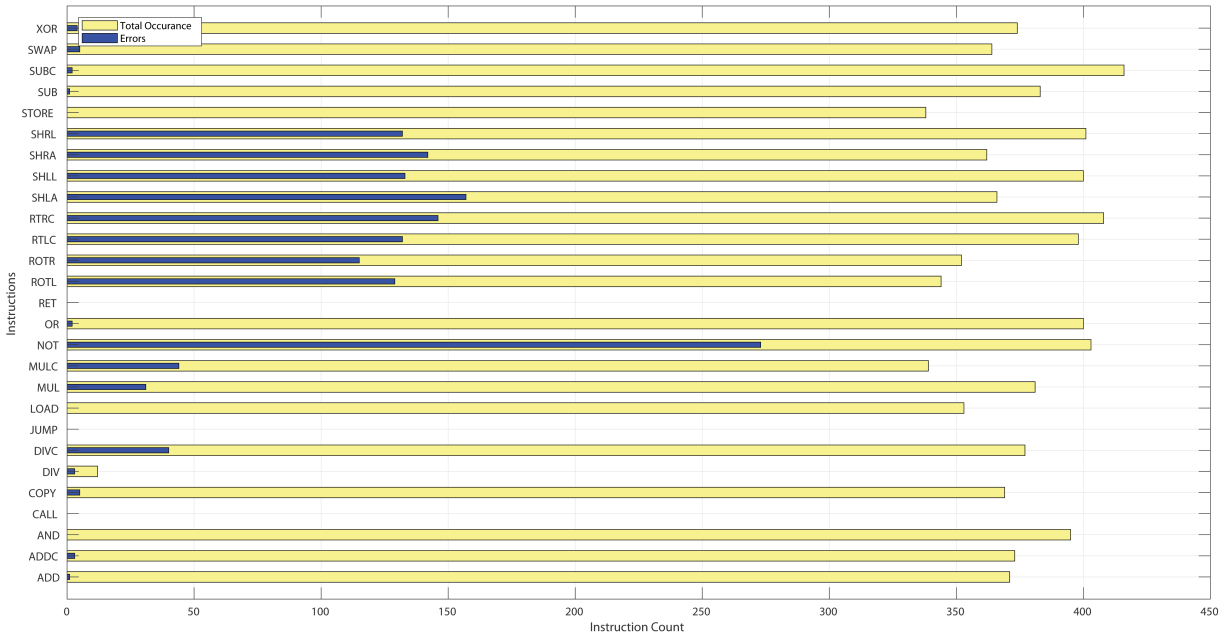

Figure V.45: Total Error count for test *T1* (mode MD) in processor *nxp*

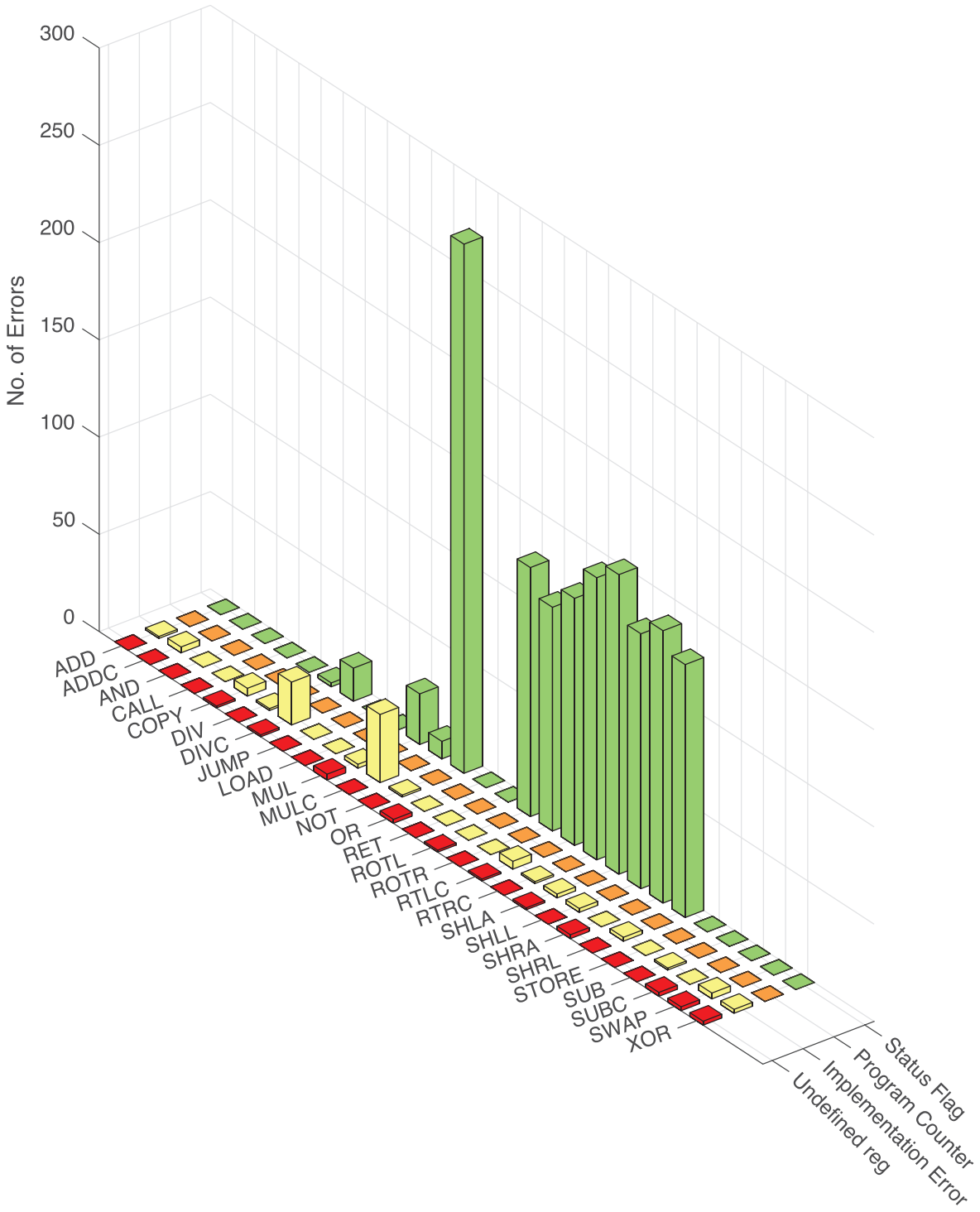

Figure V.46: Errors found in processor *nxp* while executing test *T1* (mode MD)

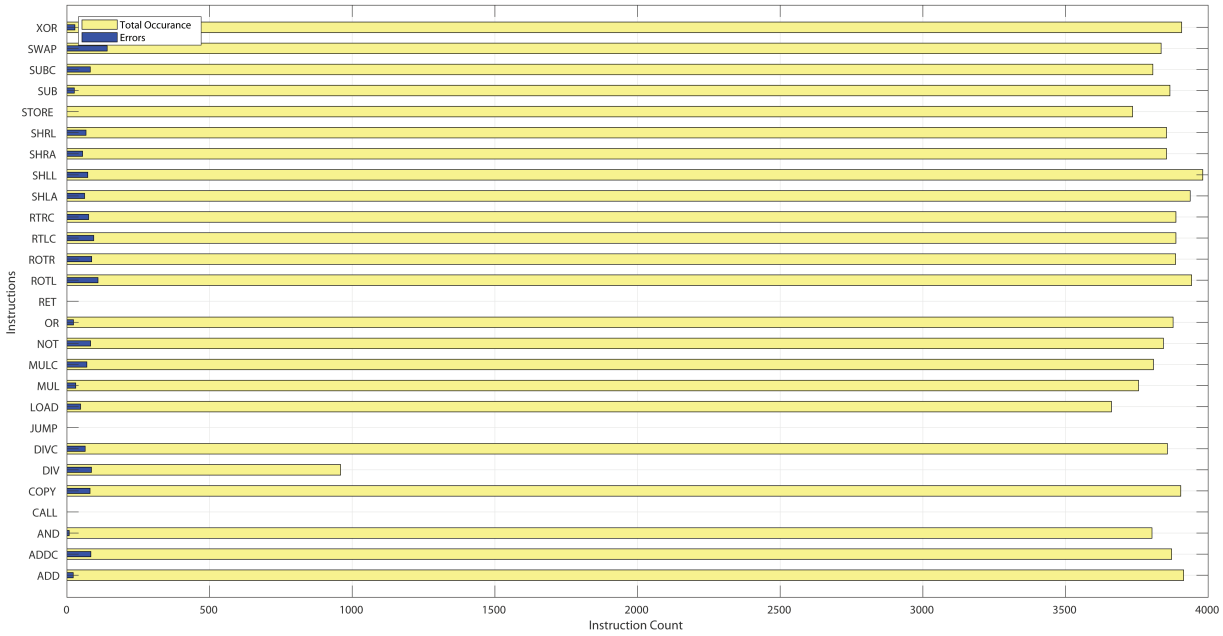

Figure V.47: Total Error count for test *T2* (mode MD) in processor *nxp*

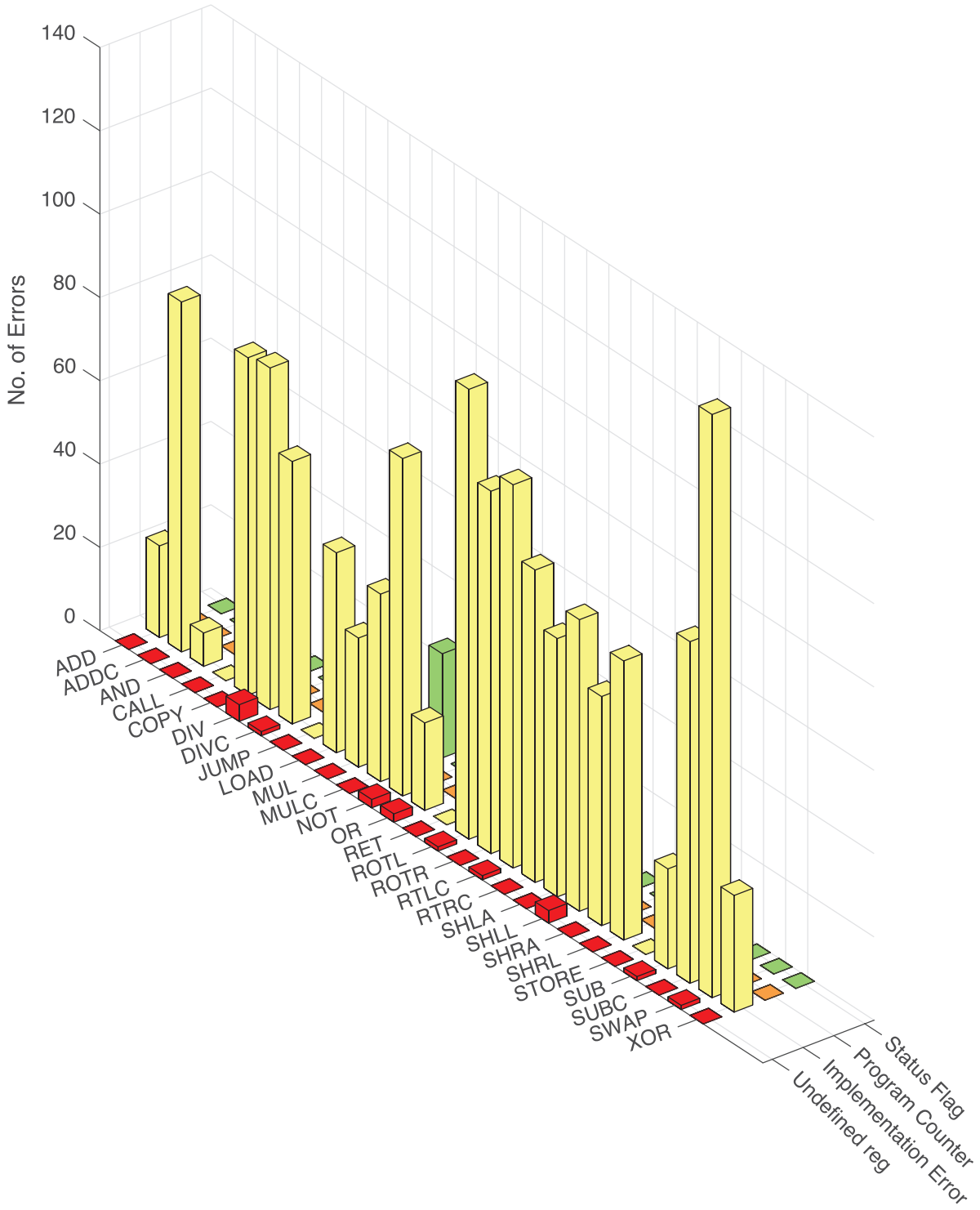

Figure V.48: Errors found in processor *nxp* while executing test *T2* (mode MD)

V.4 Processor *tfl*

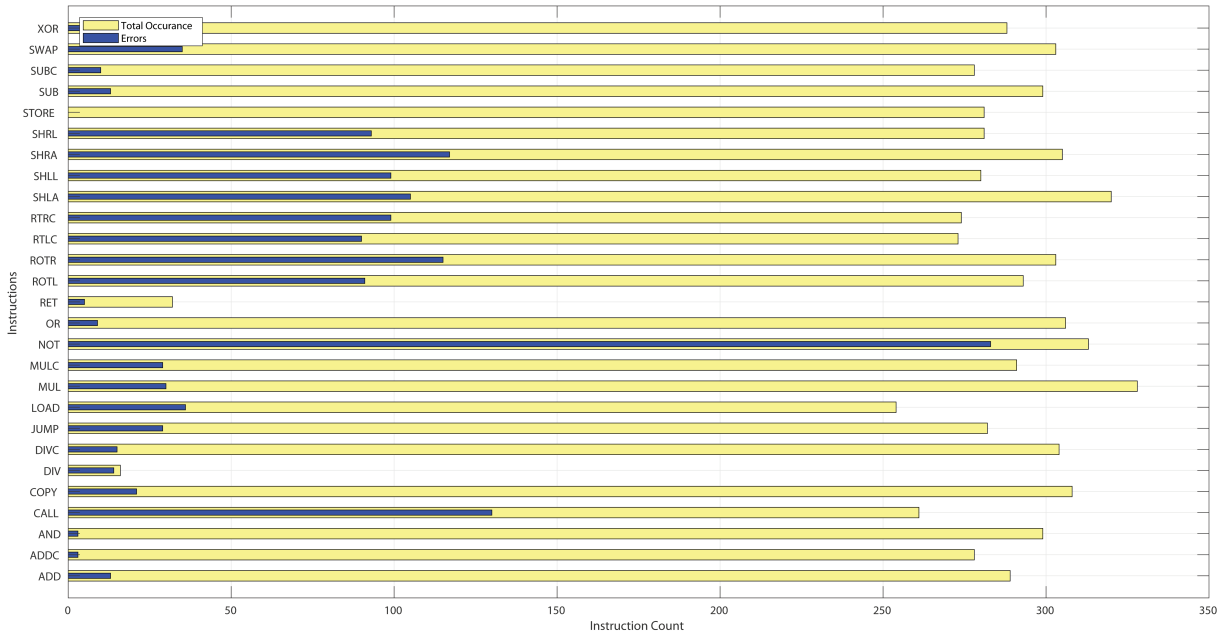

Figure V.49: Total Error count for test *T1* (mode A) in processor *tfl*

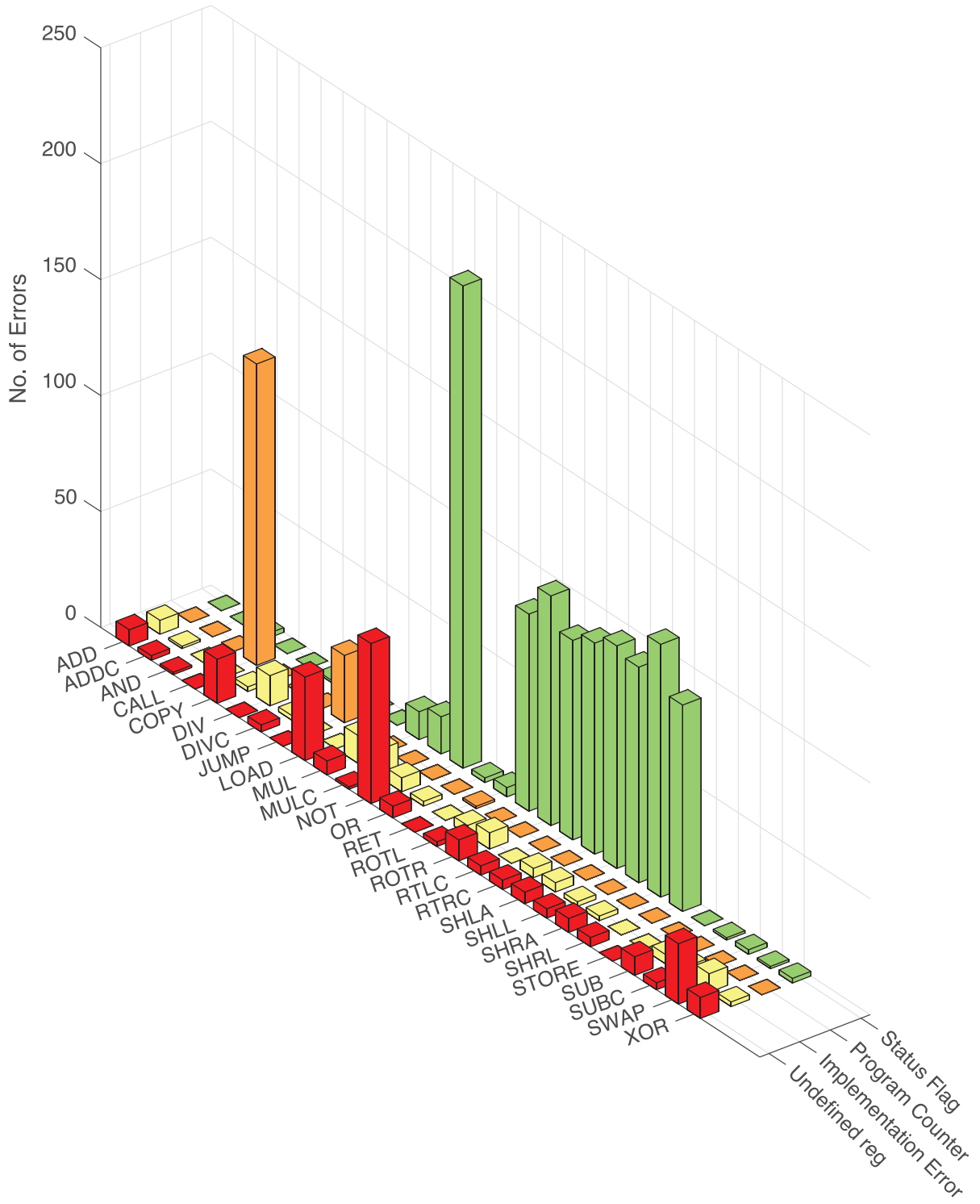

Figure V.50: Errors found in processor *tfl* while executing test *T1* (mode A)

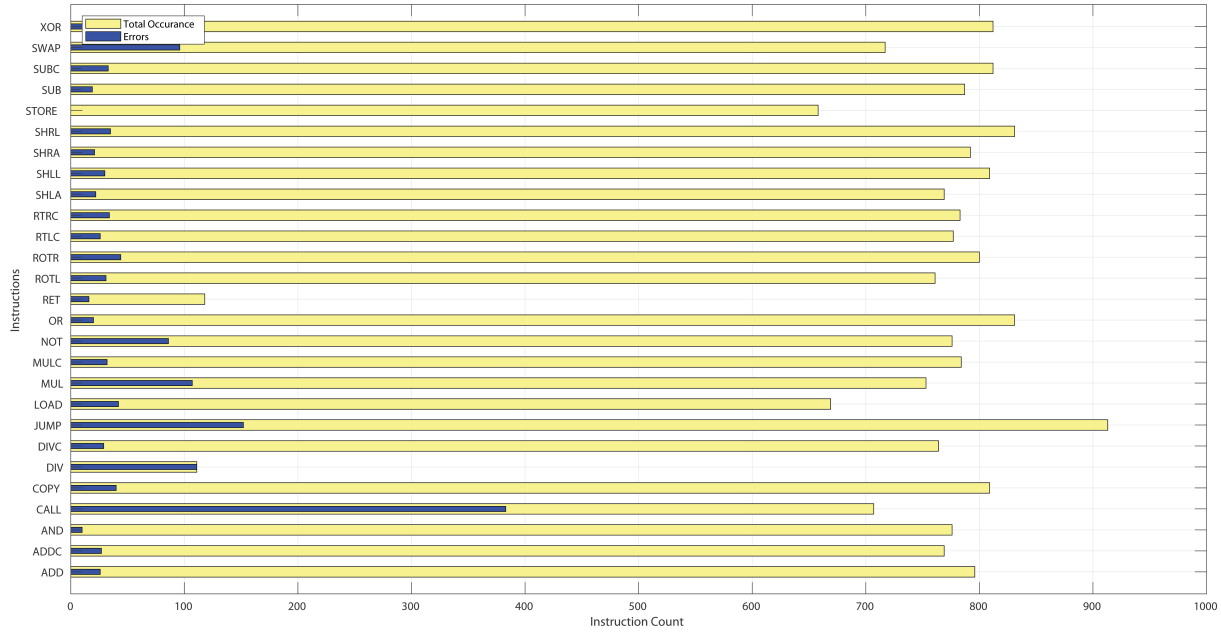

Figure V.51: Total Error count for test *T2* (mode A) in processor *tfl*

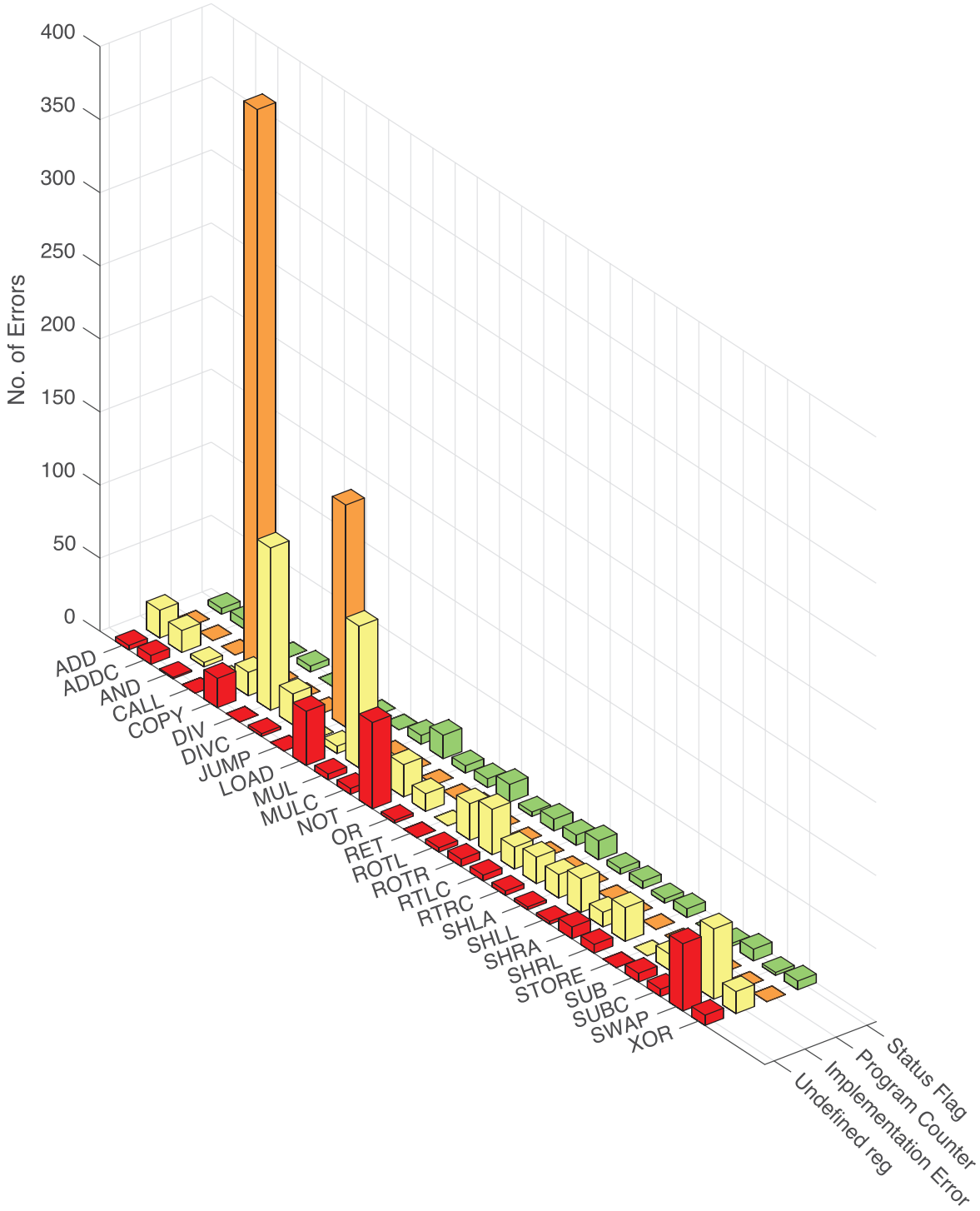

Figure V.52: Errors found in processor *tfl* while executing test *T2* (mode A)

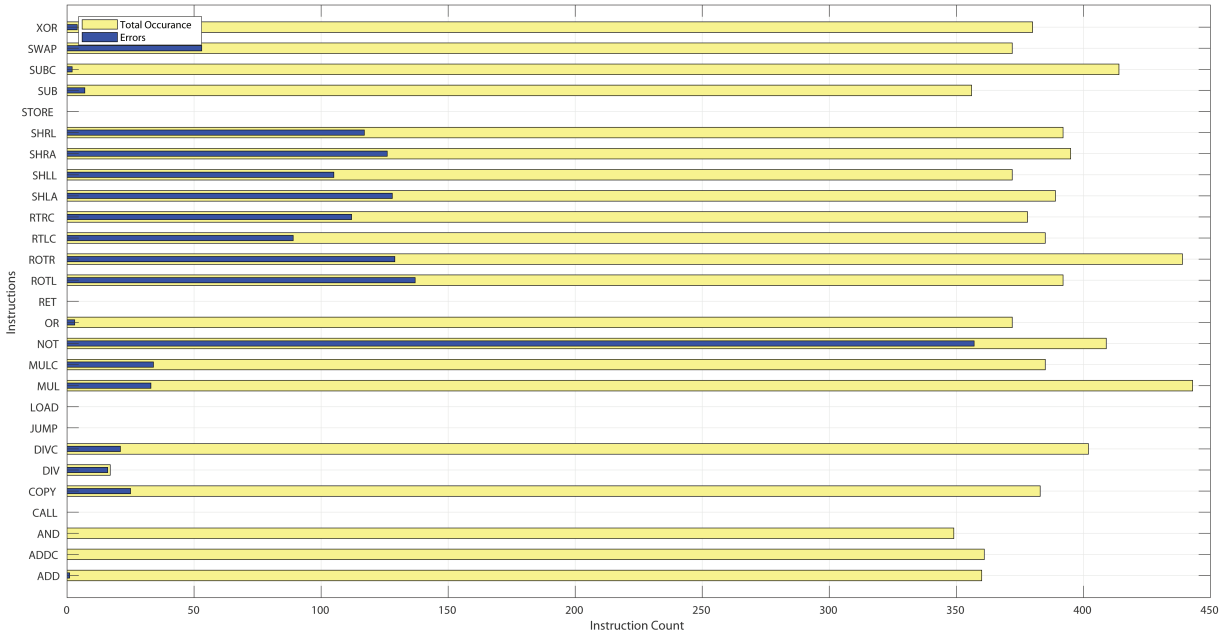

Figure V.53: Total Error count for test *T1* (mode M) in processor *tfl*

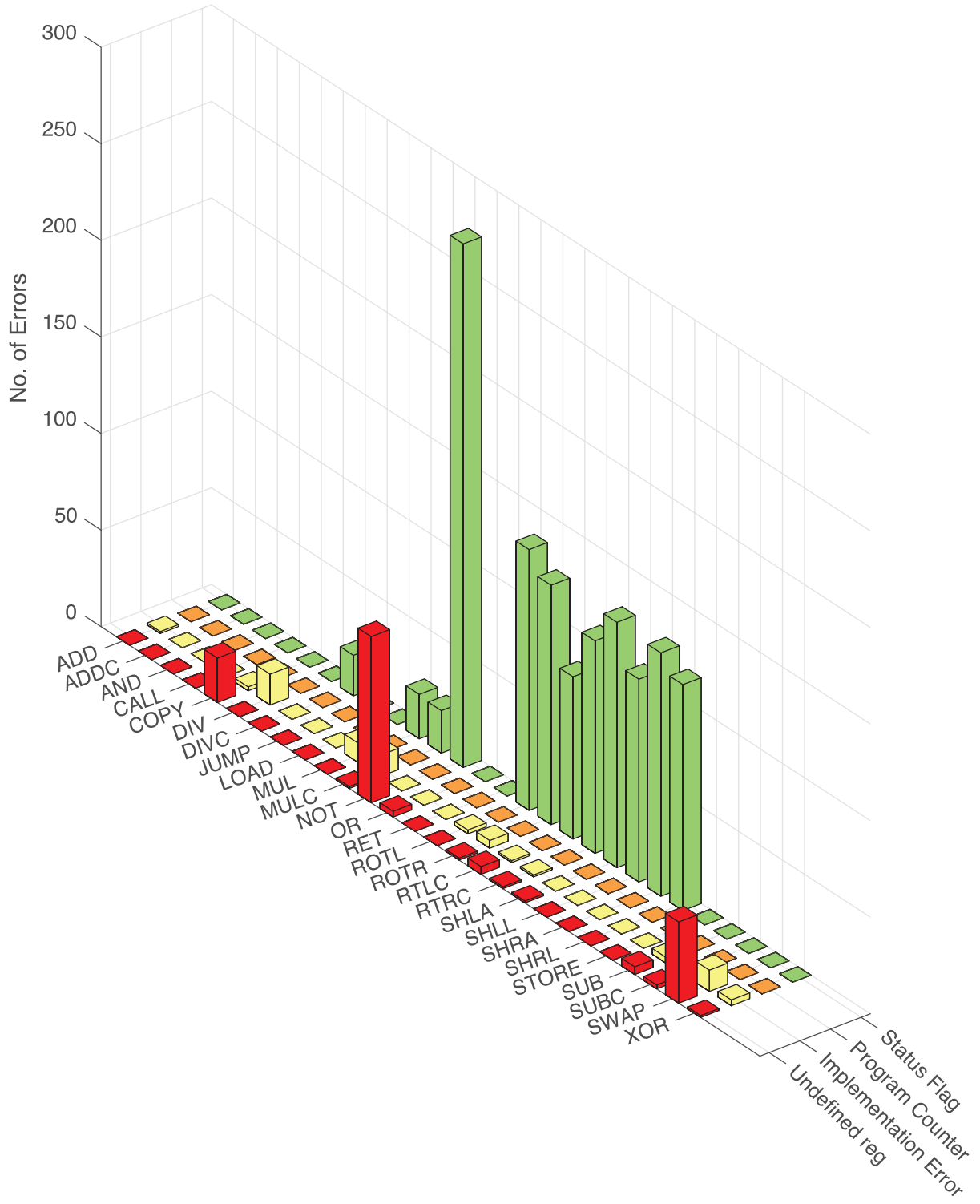

Figure V.54: Errors found in processor *tfl* while executing test *T1* (mode M)

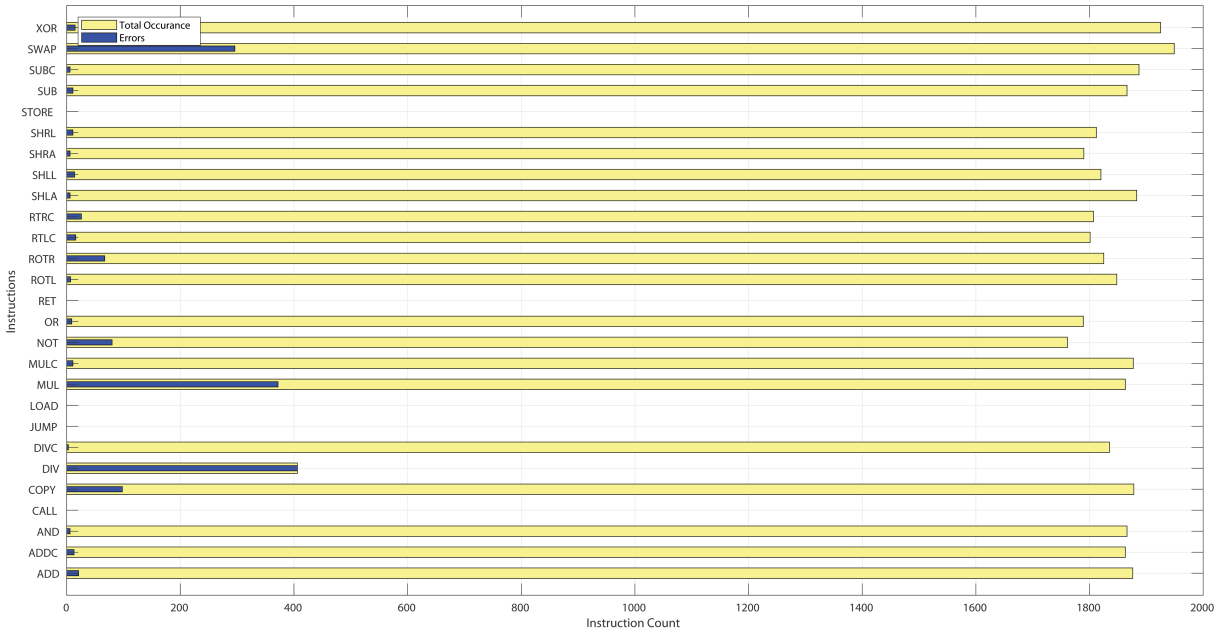

Figure V.55: Total Error count for test *T2* (mode M) in processor *tfl*

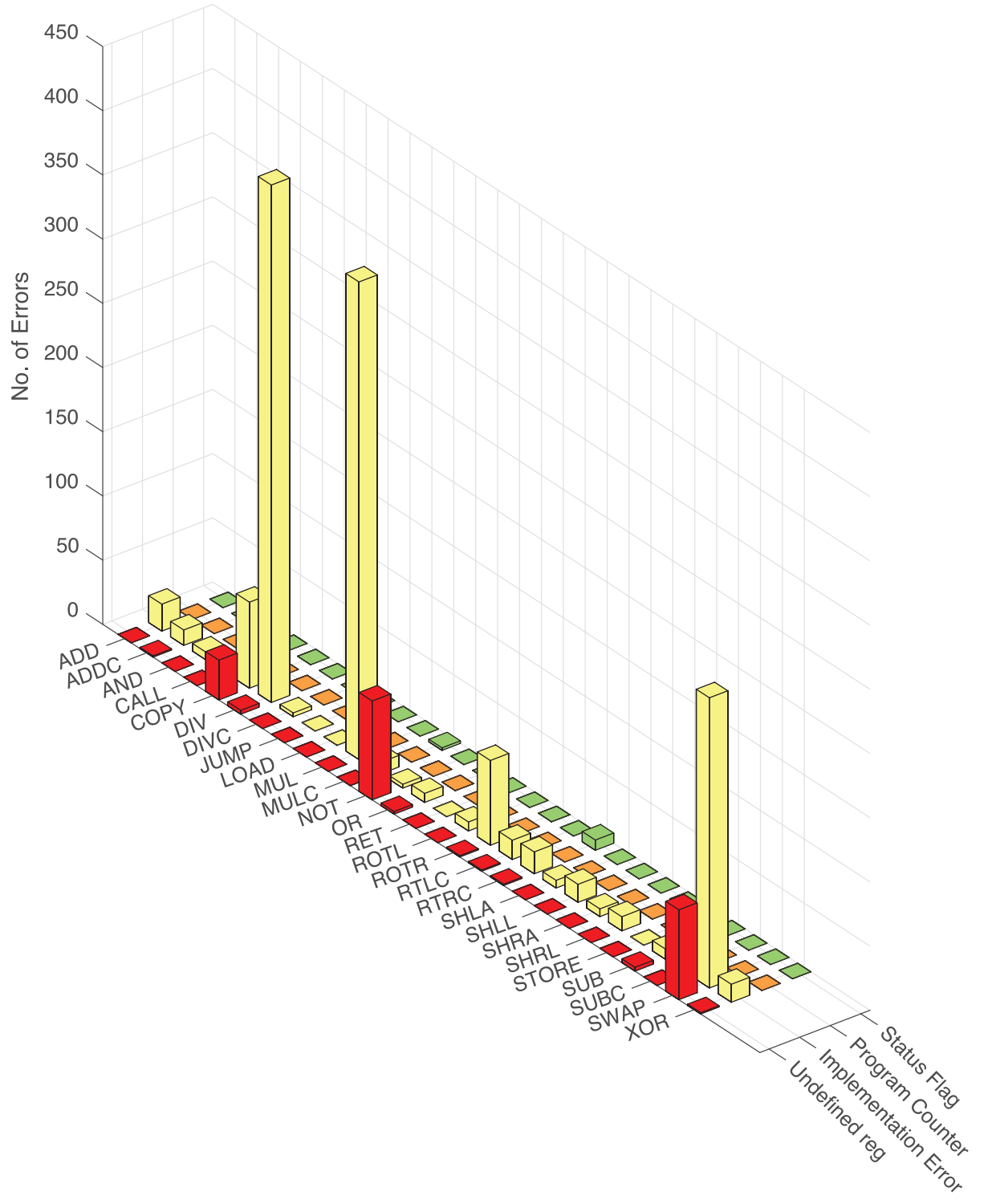

Figure V.56: Errors found in processor *tfl* while executing test *T2* (mode M)

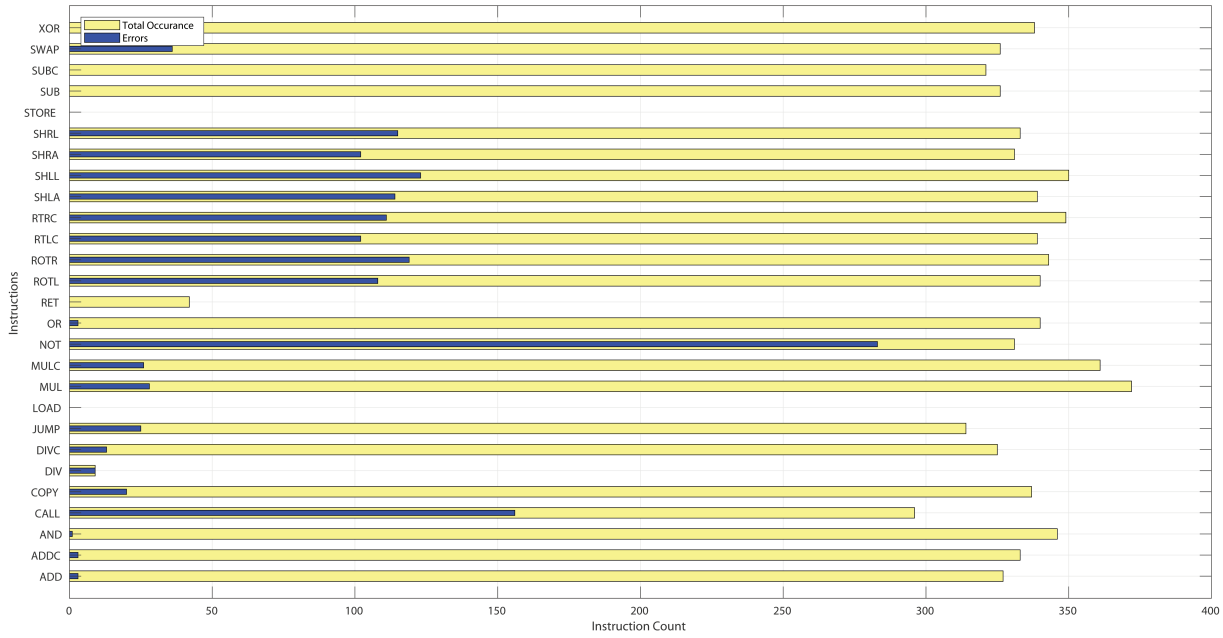

Figure V.57: Total Error count for test *T1* (mode MB) in processor *tfl*

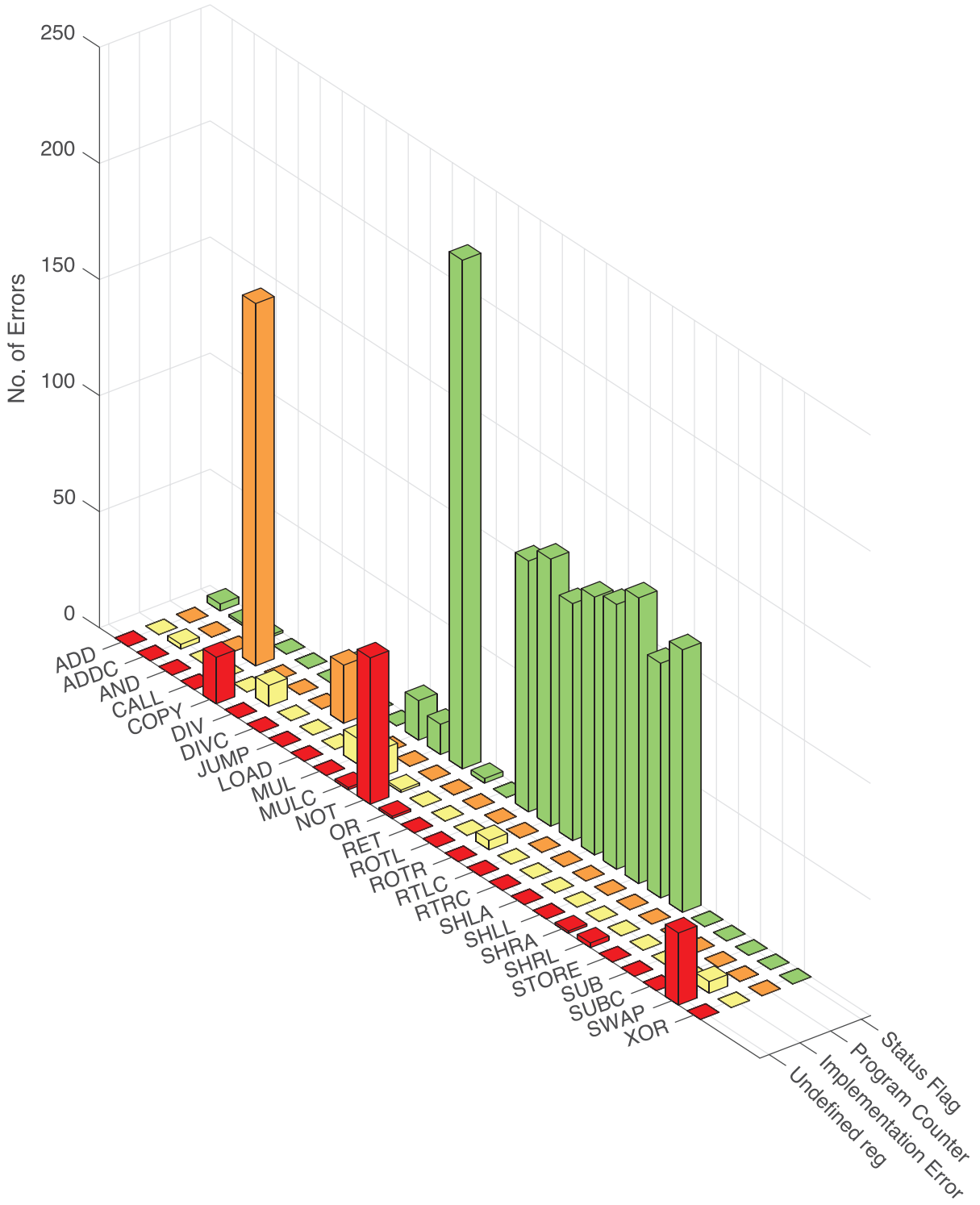

Figure V.58: Errors found in processor *tfl* while executing test *T1* (mode MB)

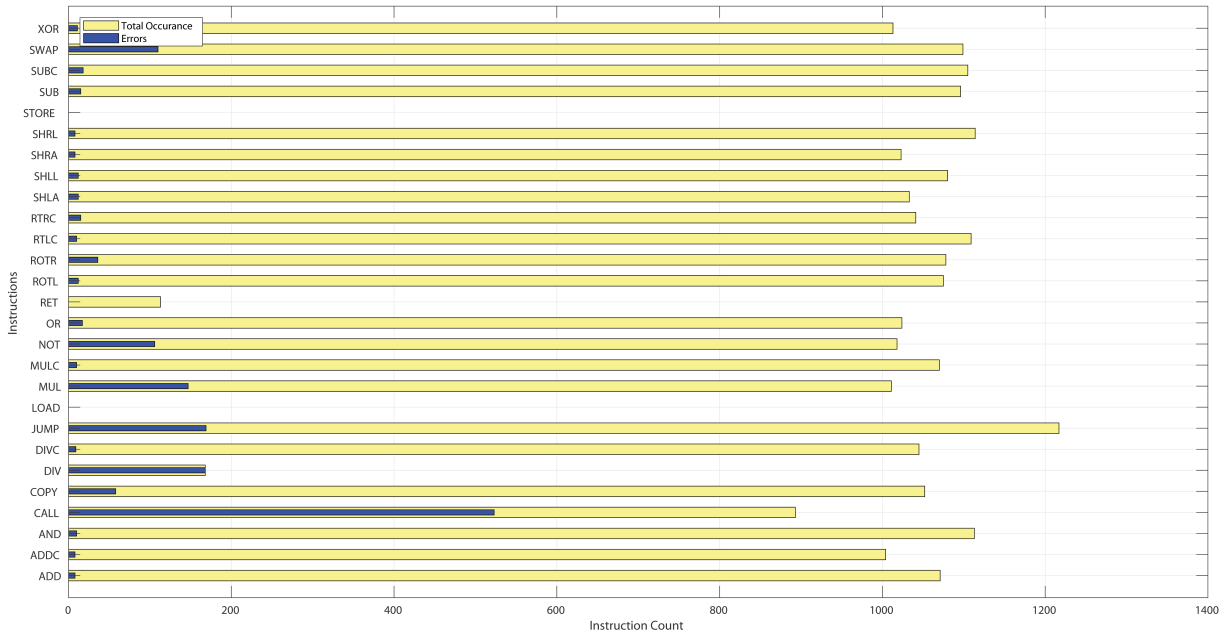

Figure V.59: Total Error count for test *T2* (mode MB) in processor *tfl*

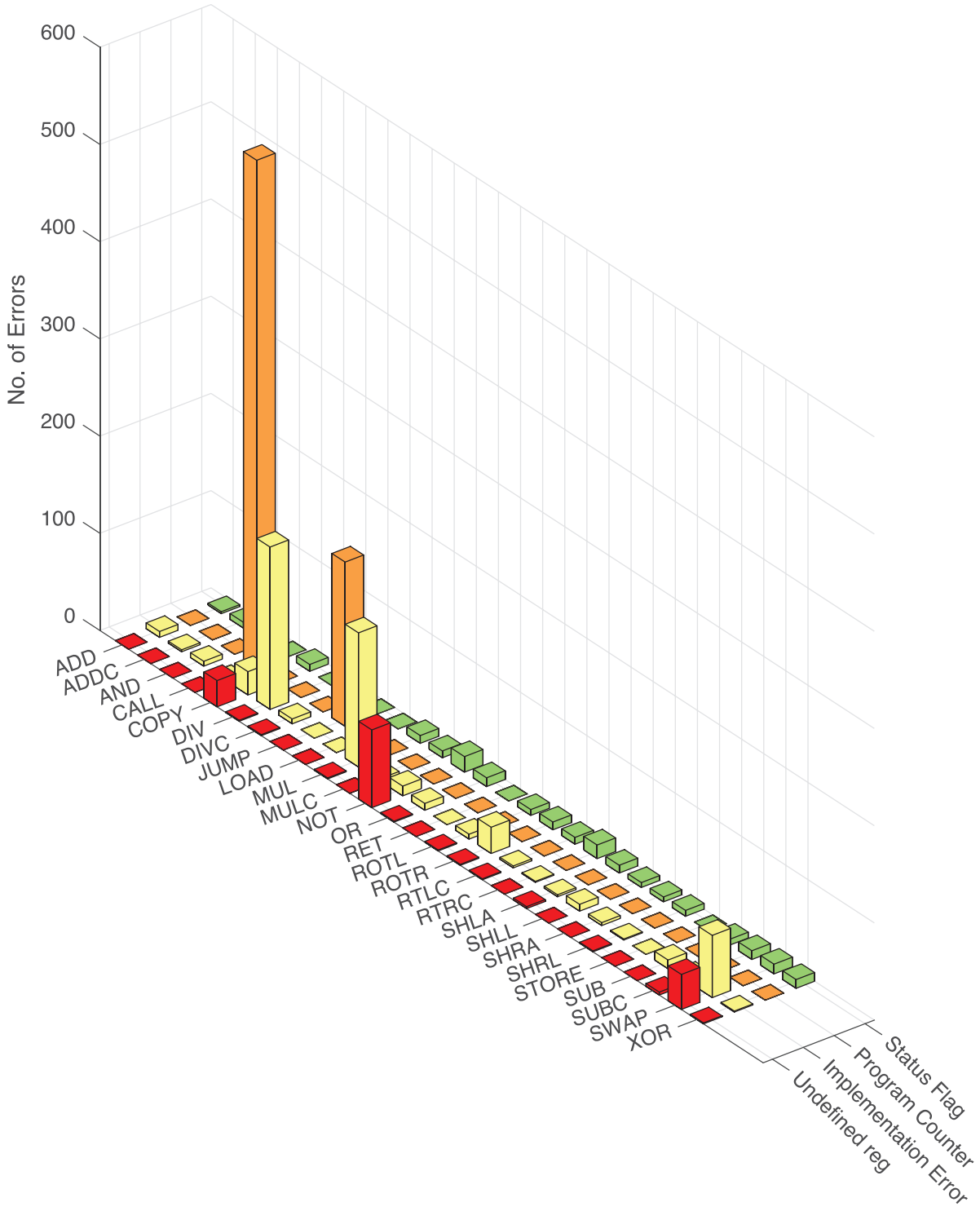

Figure V.60: Errors found in processor *tfl* while executing test *T2* (mode MB)

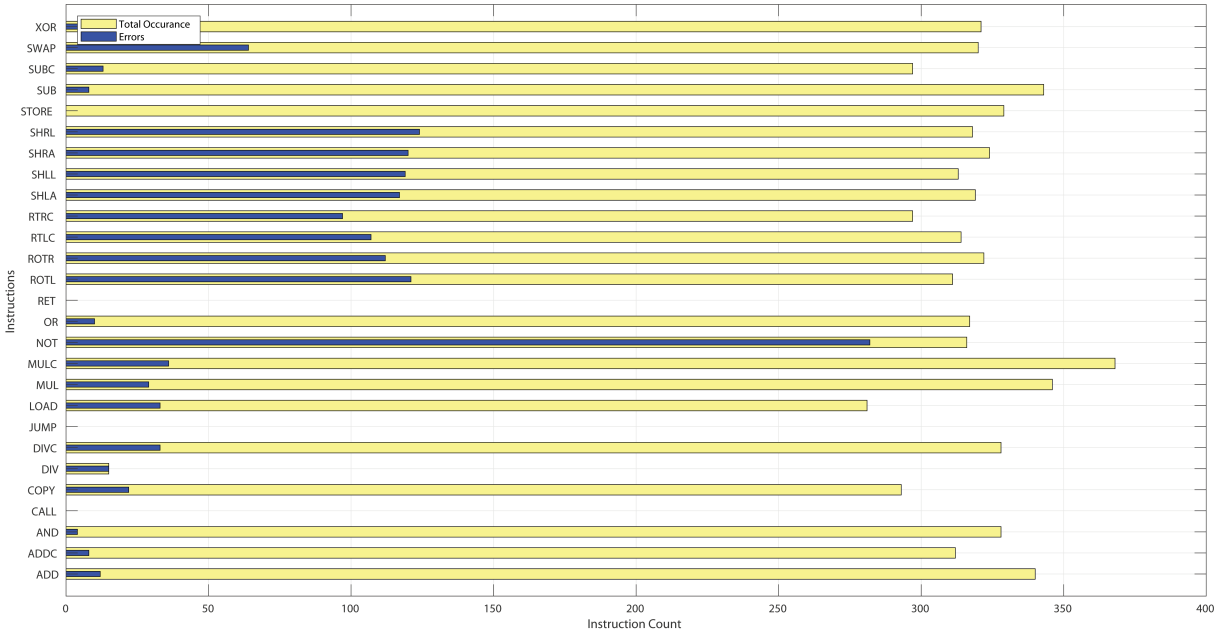

Figure V.61: Total Error count for test *T1* (mode MD) in processor *tfl*

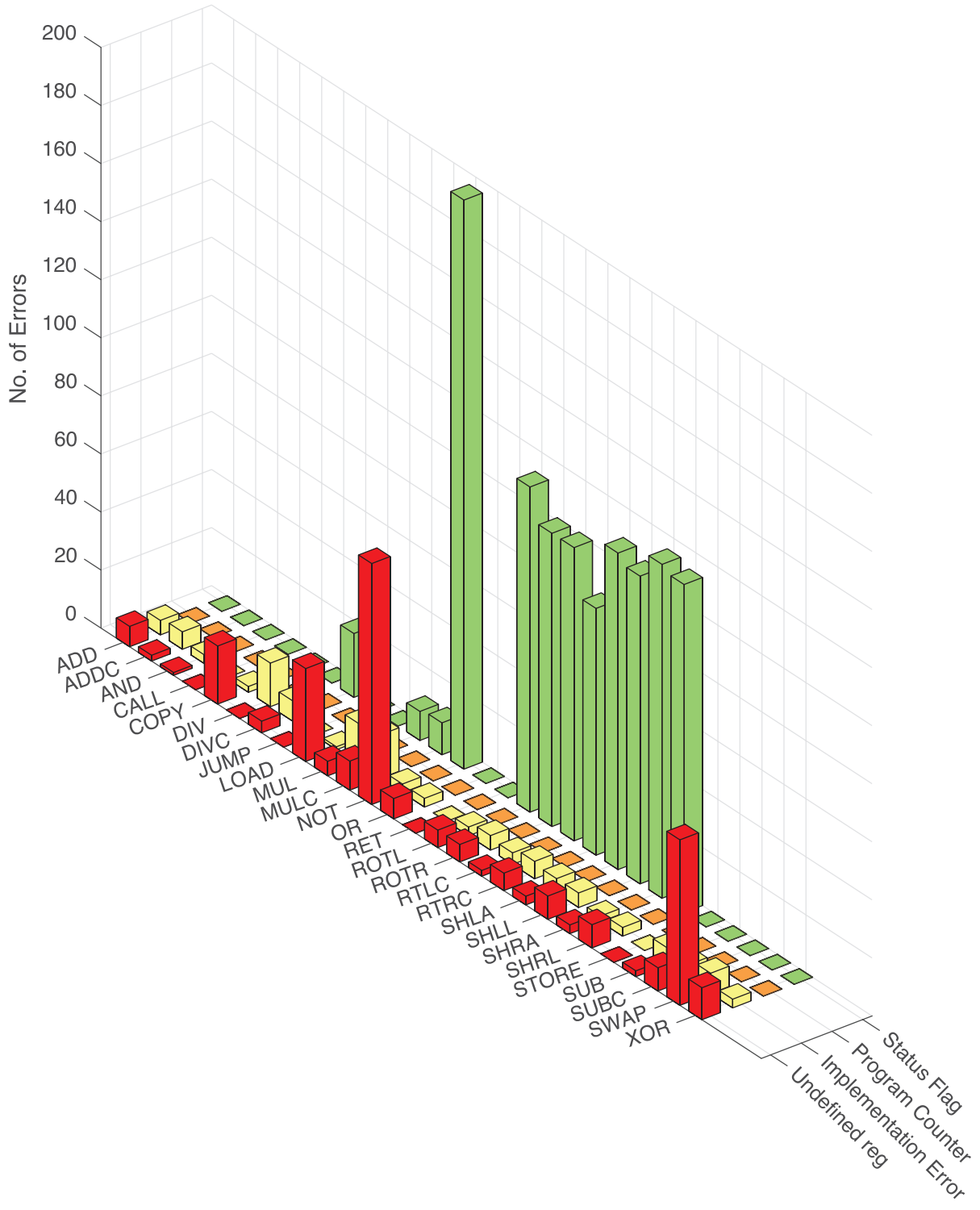

Figure V.62: Errors found in processor *tfl* while executing test *T1* (mode MD)

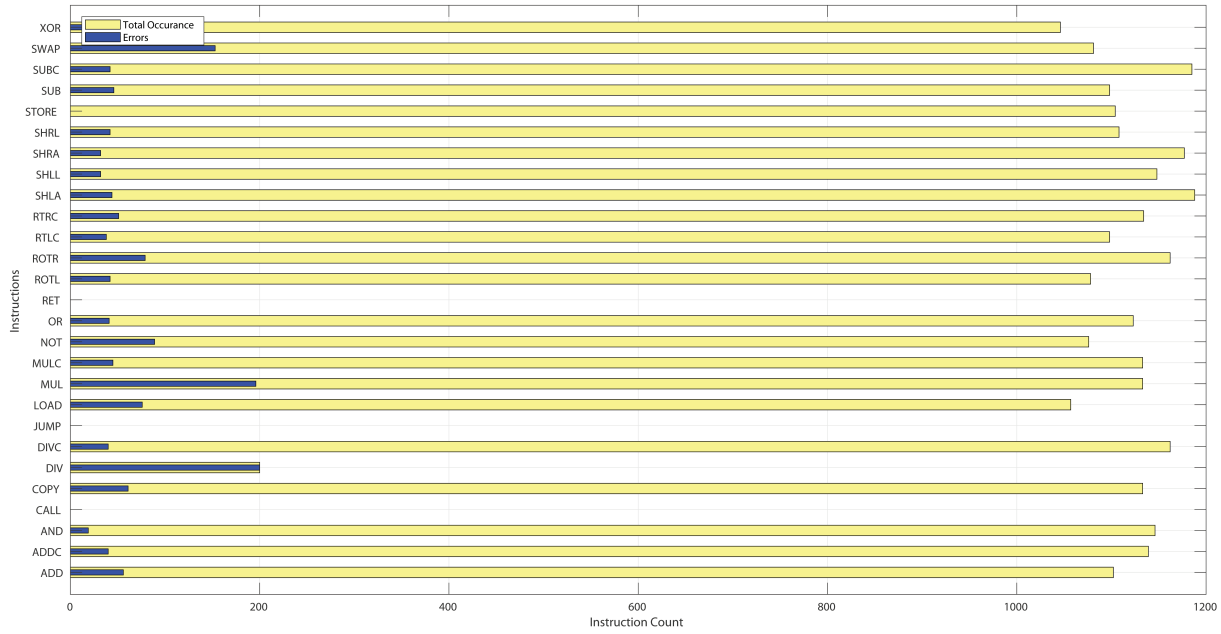

Figure V.63: Total Error count for test *T2* (mode MD) in processor *tfl*

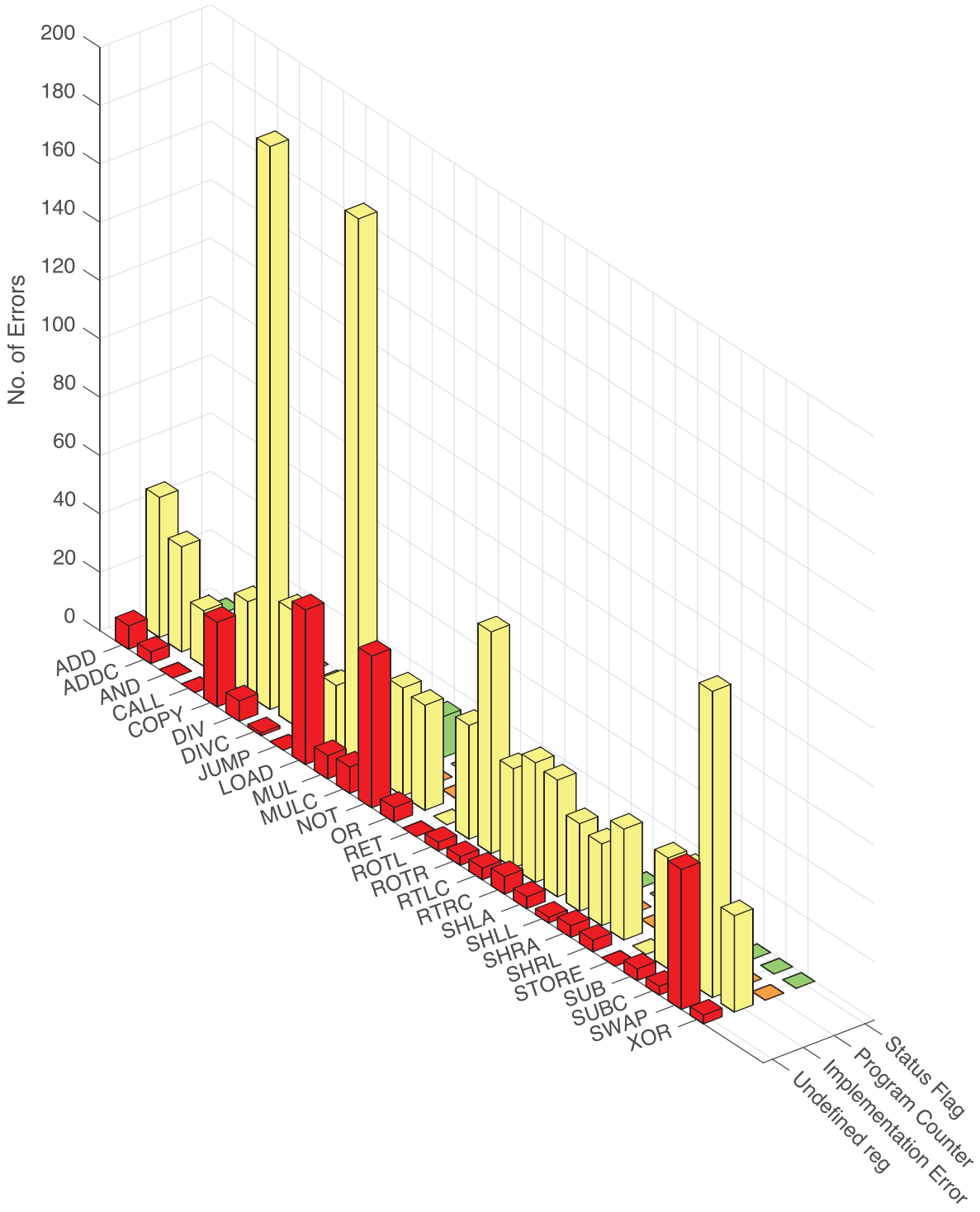

Figure V.64: Errors found in processor *tfl* while executing test *T2* (mode MD)

V.5 Processor *sxs*

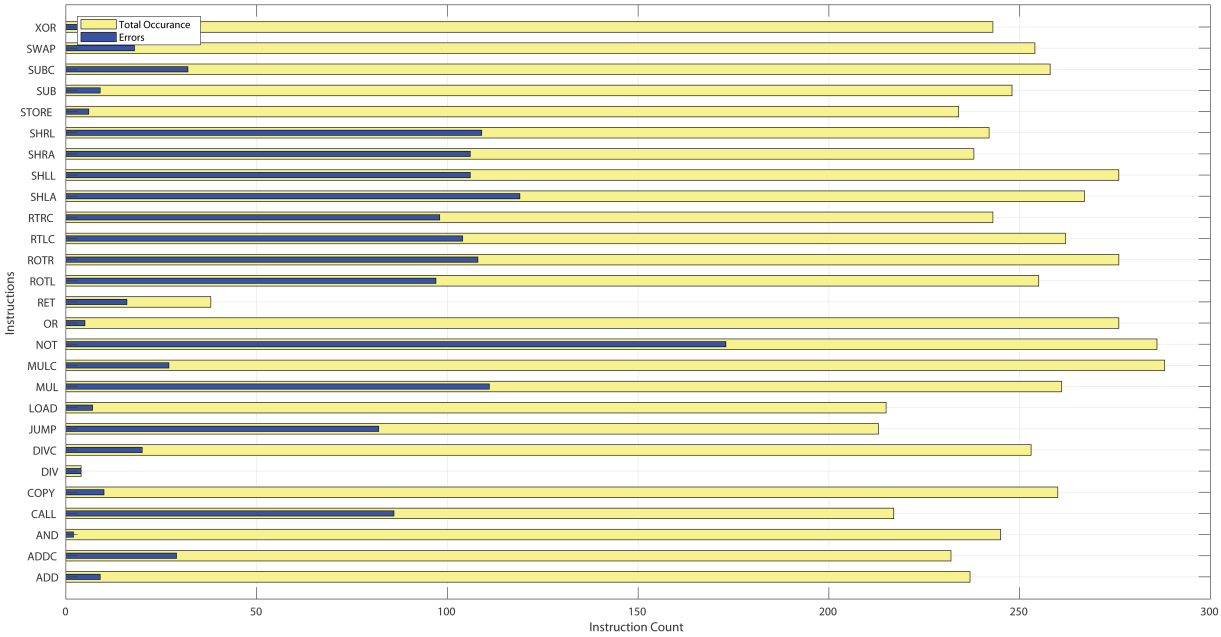

Figure V.65: Total Error count for test *T1* (mode A) in processor *sxs*

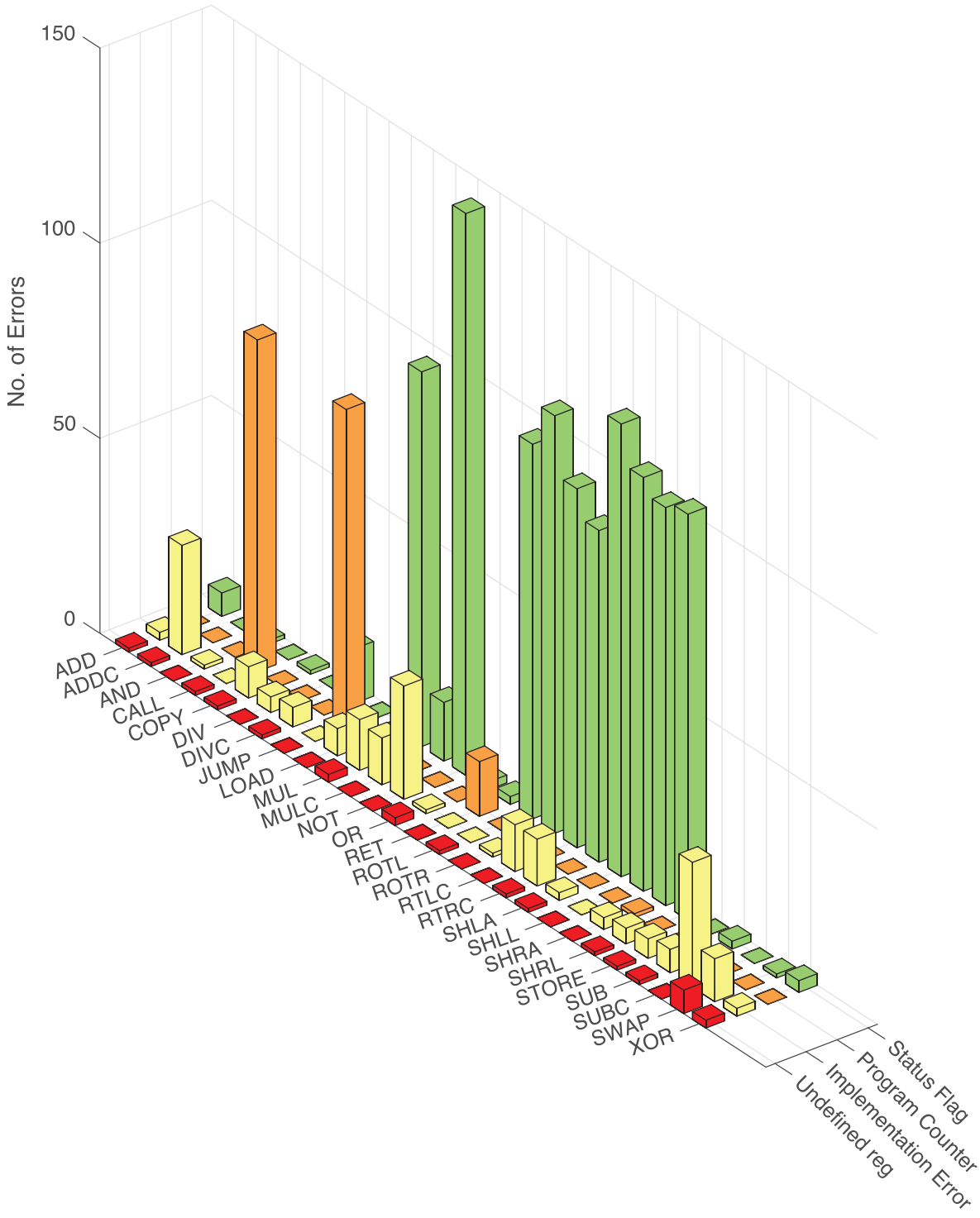

Figure V.66: Errors found in processor *sxs* while executing test *T1* (mode A)

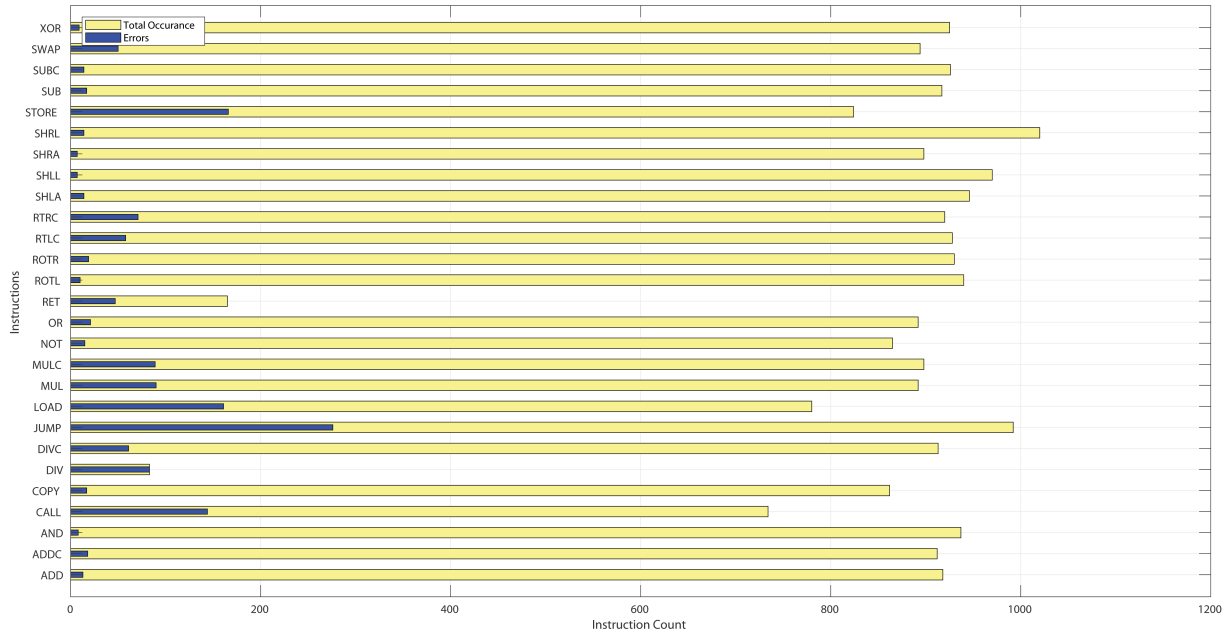

Figure V.67: Total Error count for test *T2* (mode A) in processor *sxs*

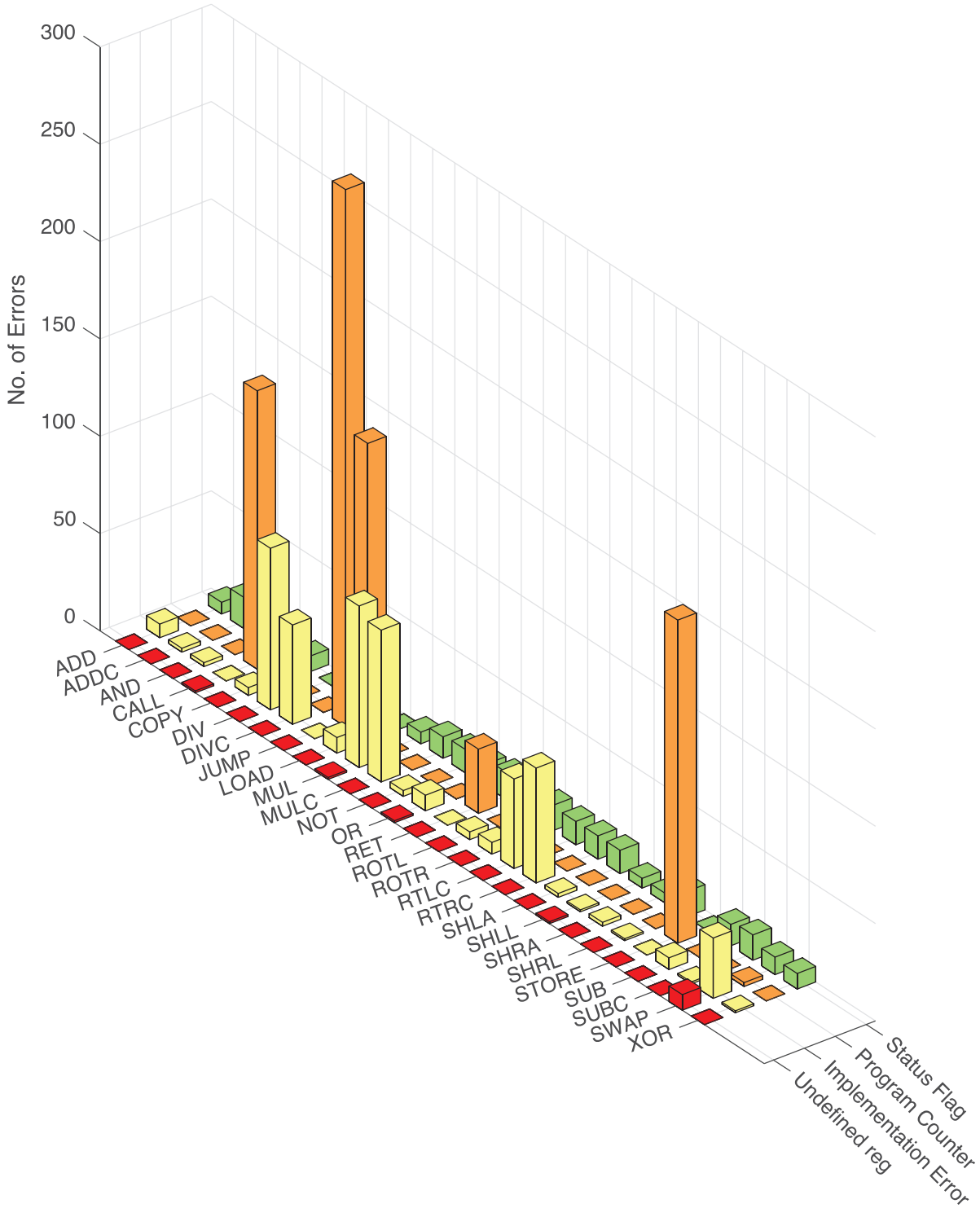

Figure V.68: Errors found in processor *sxs* while executing test *T2* (mode A)

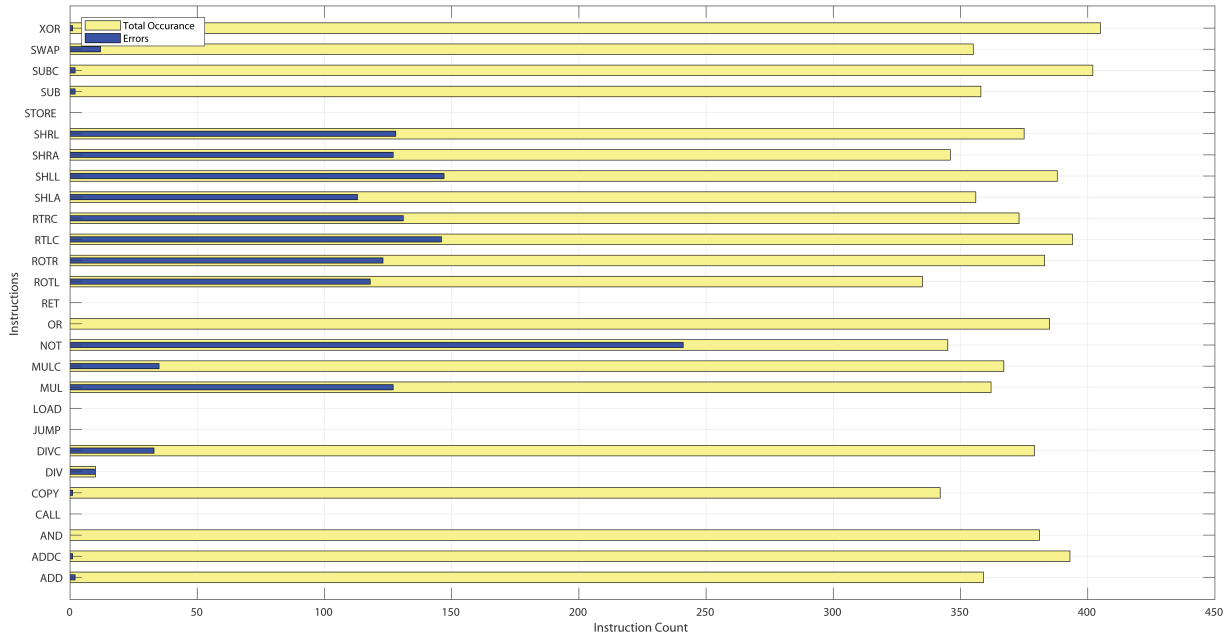

Figure V.69: Total Error count for test *T1* (mode M) in processor *sxs*

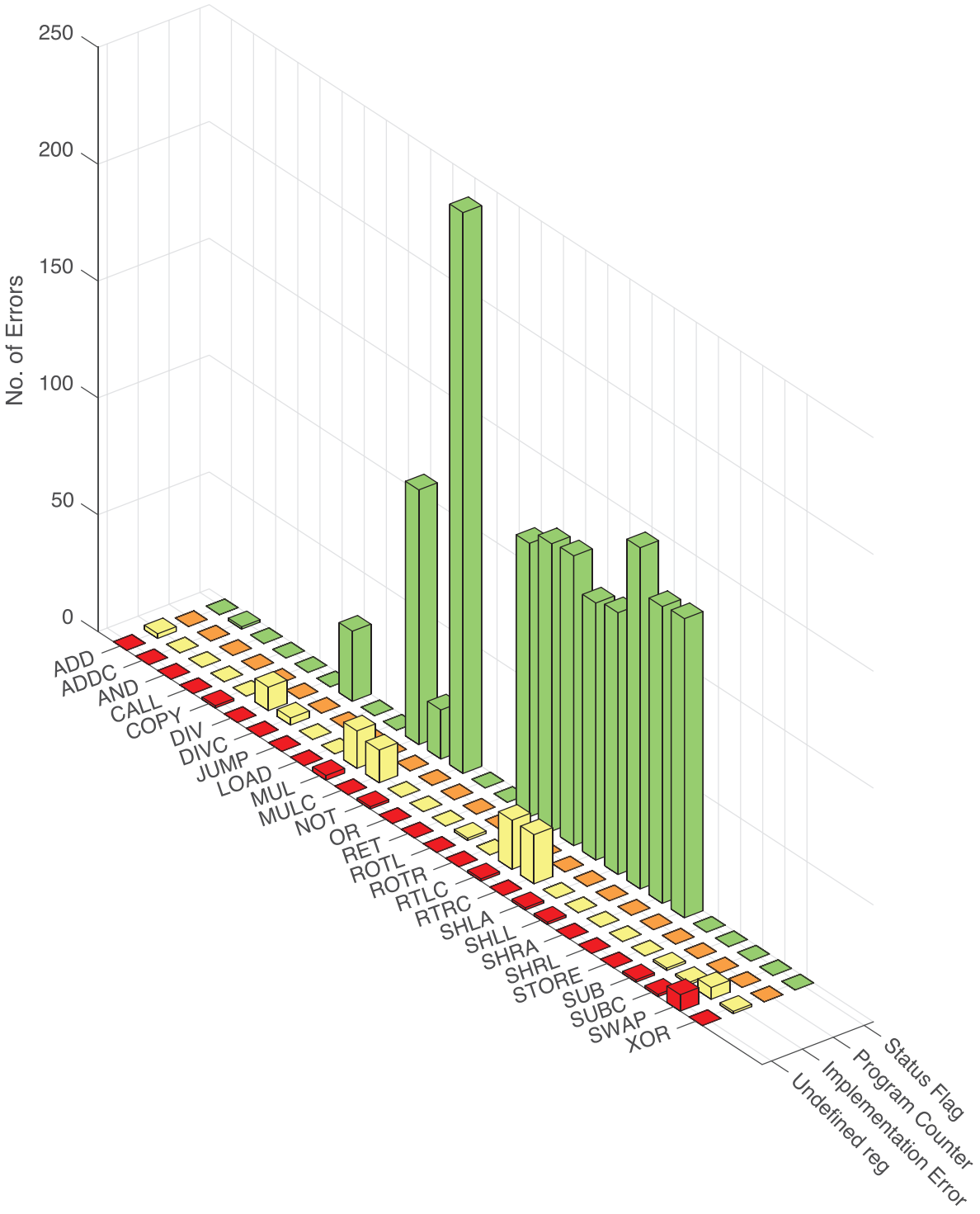

Figure V.70: Errors found in processor *sxs* while executing test *T1* (mode M)

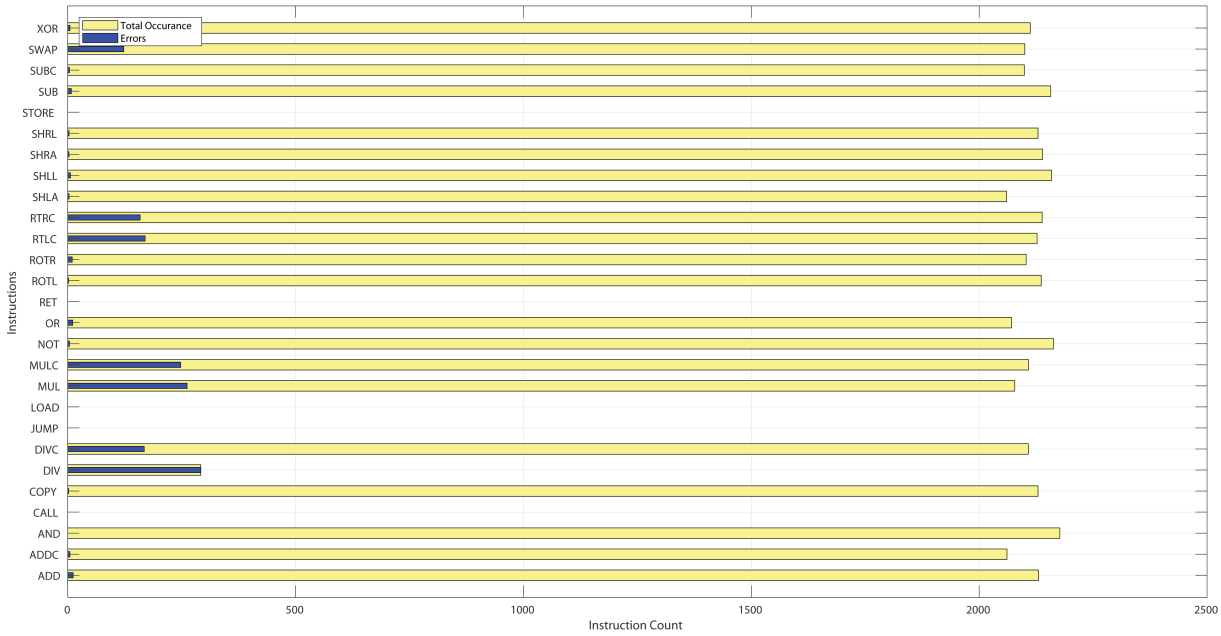

Figure V.71: Total Error count for test *T2* (mode M) in processor *sxs*

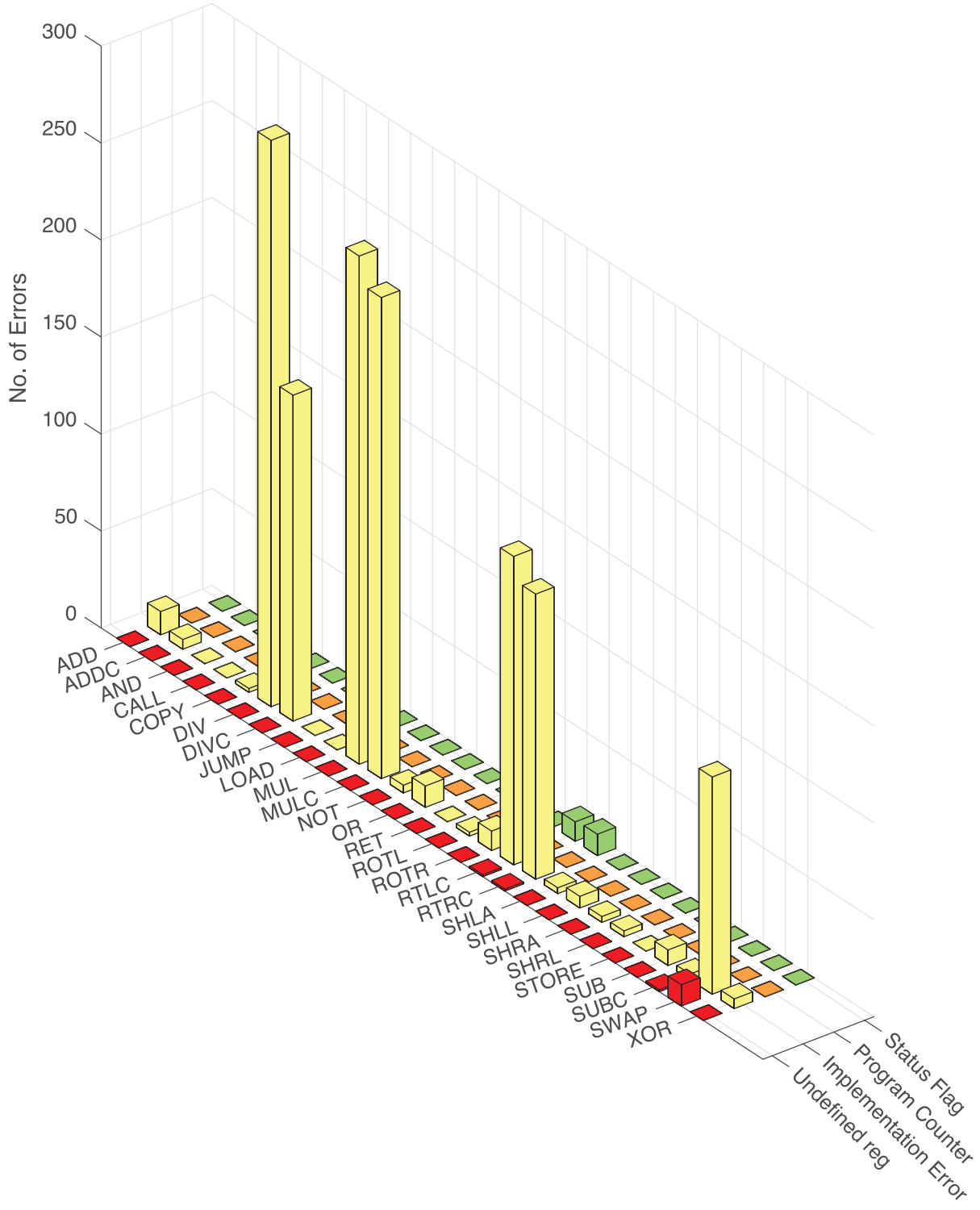

Figure V.72: Errors found in processor *sxs* while executing test *T2* (mode M)

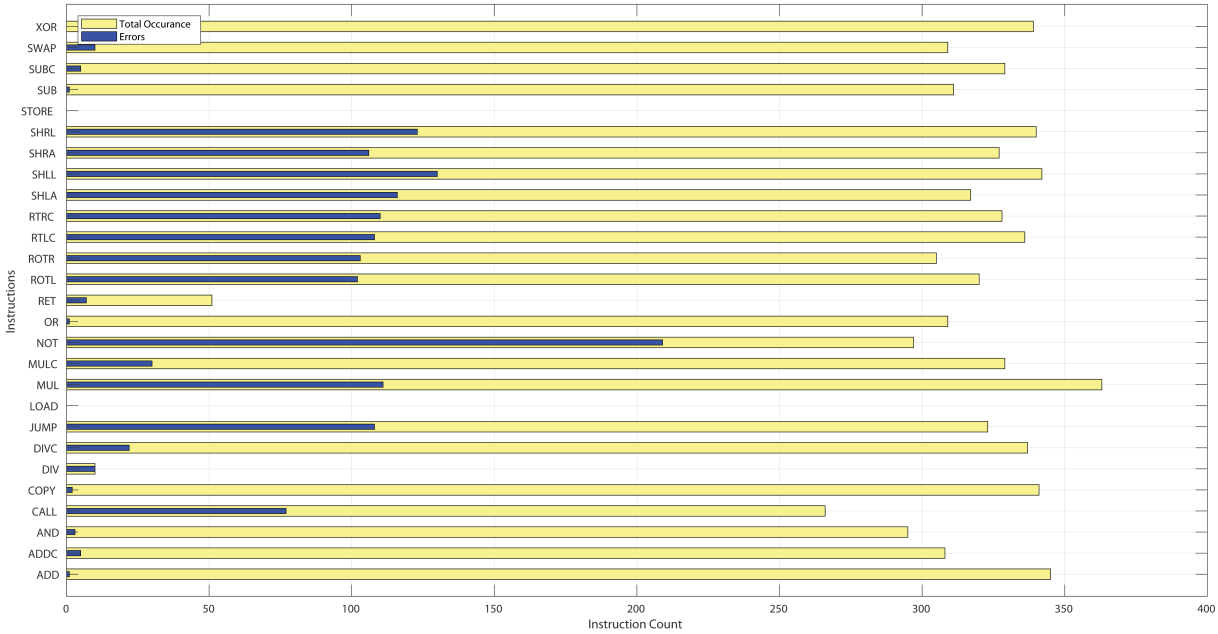

Figure V.73: Total Error count for test *T1* (mode MB) in processor *sxs*

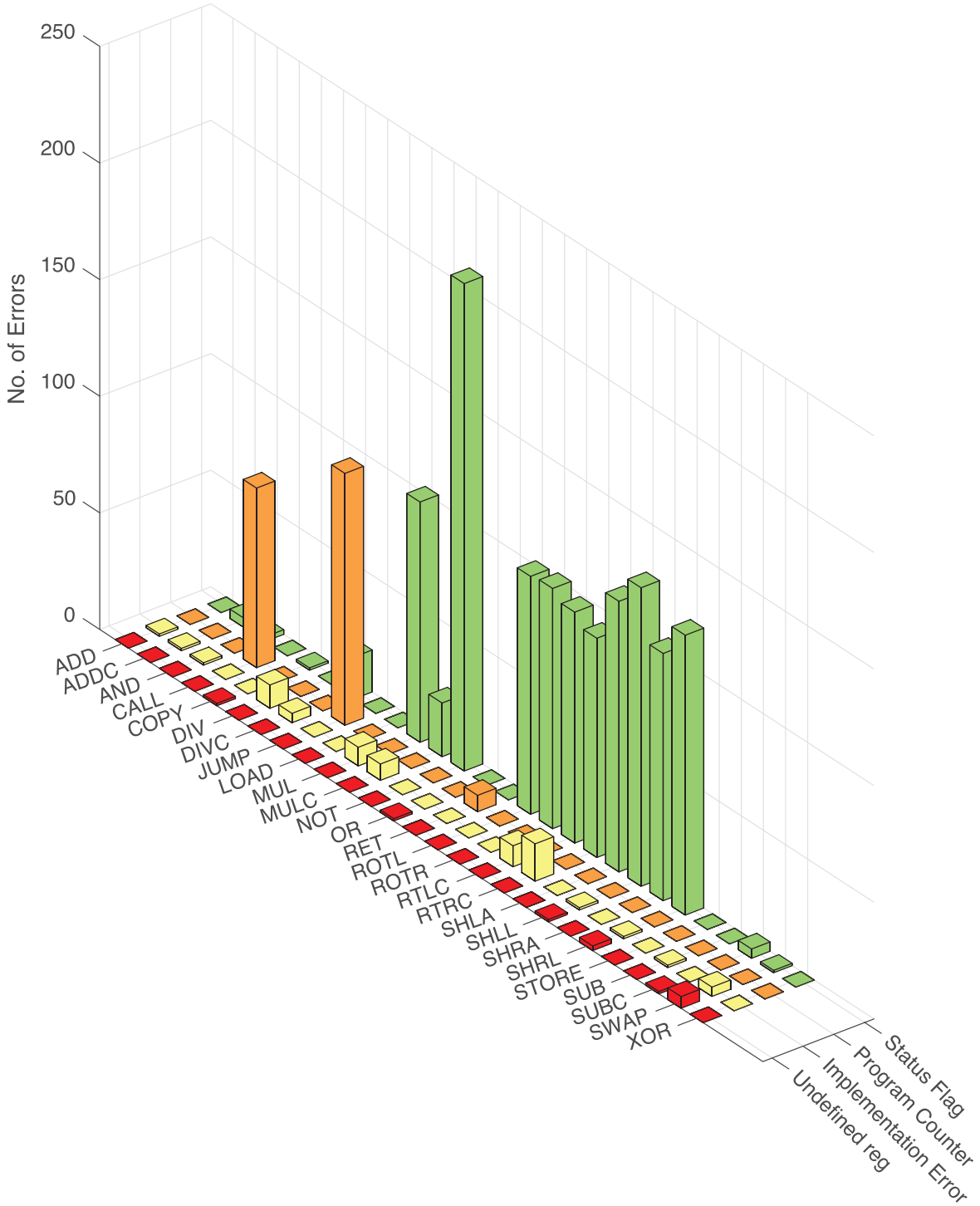

Figure V.74: Errors found in processor *sxs* while executing test *T1* (mode MB)

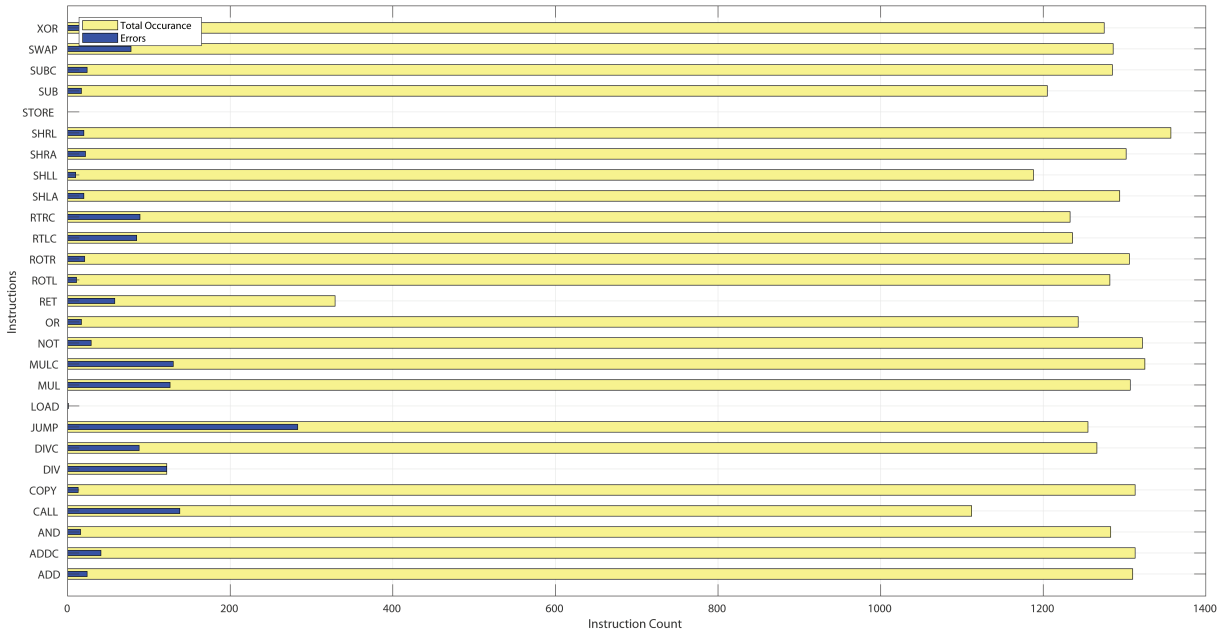

Figure V.75: Total Error count for test *T2* (mode MB) in processor *sxs*

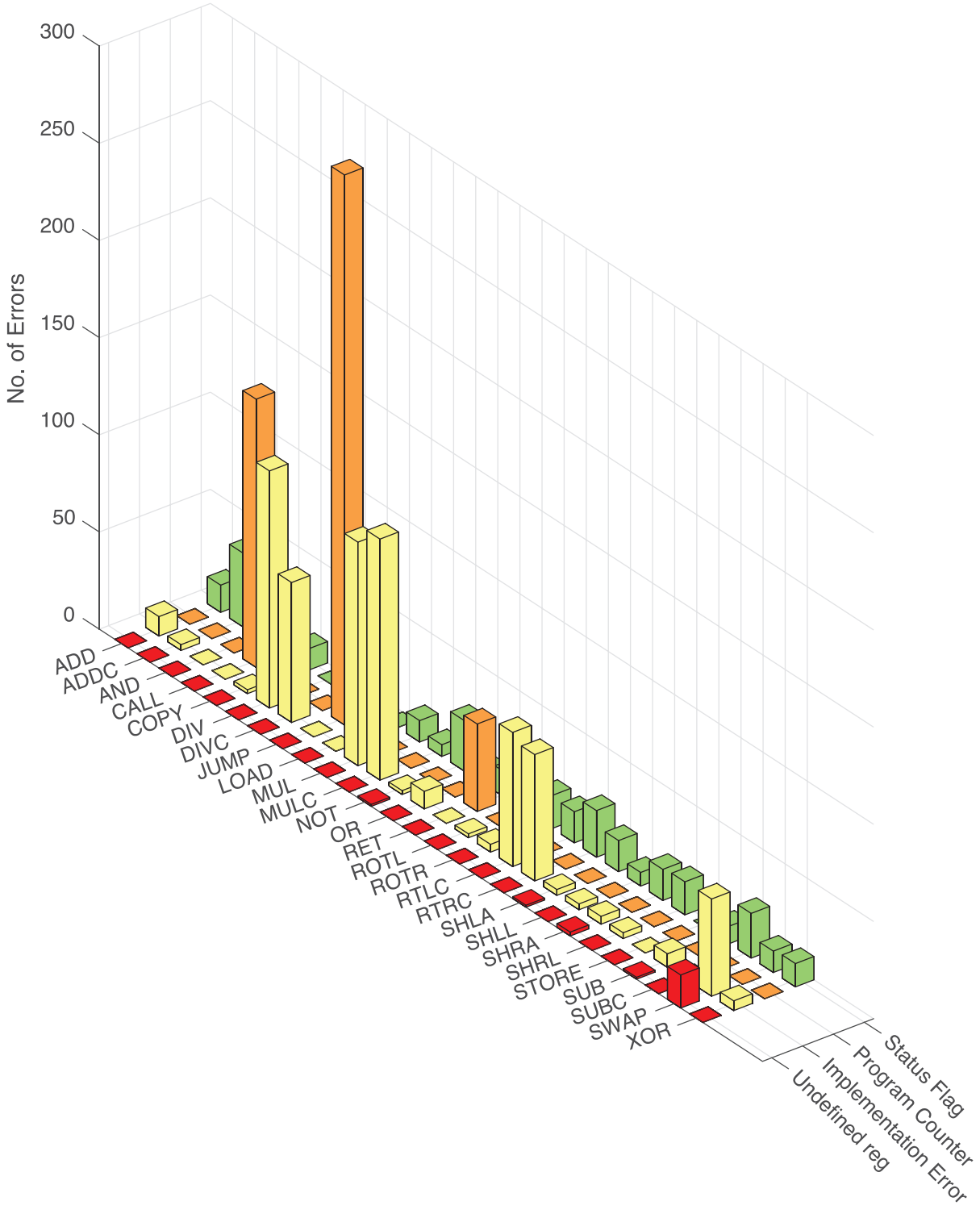

Figure V.76: Errors found in processor *sxs* while executing test *T2* (mode MB)

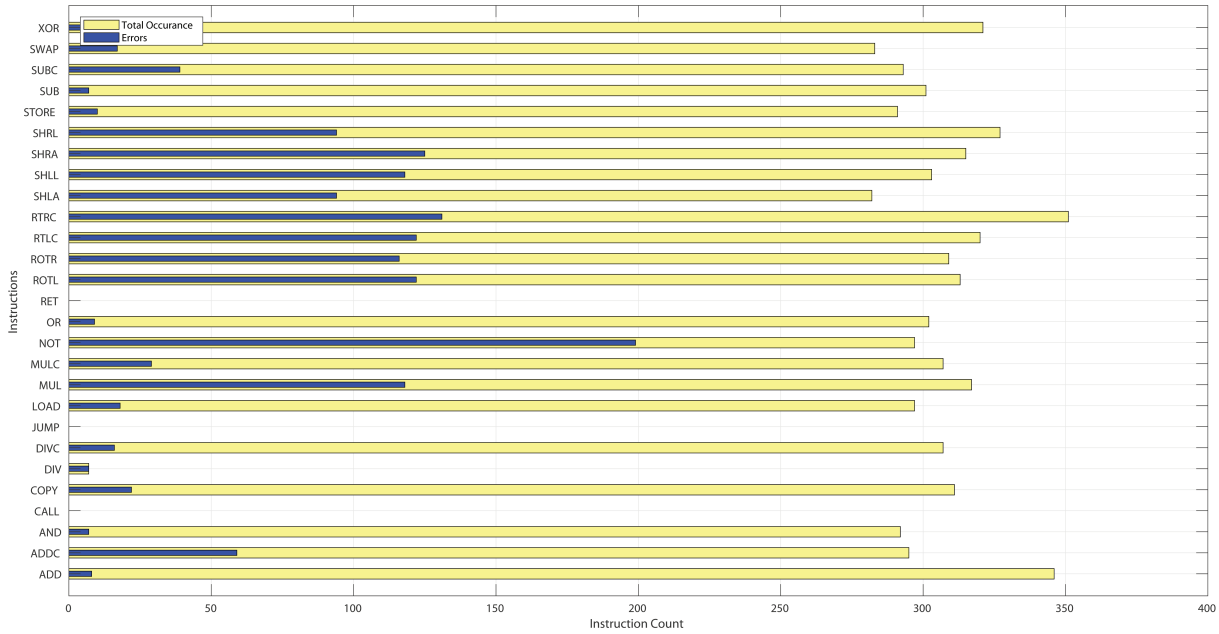

Figure V.77: Total Error count for test *T1* (mode MD) in processor *sxs*

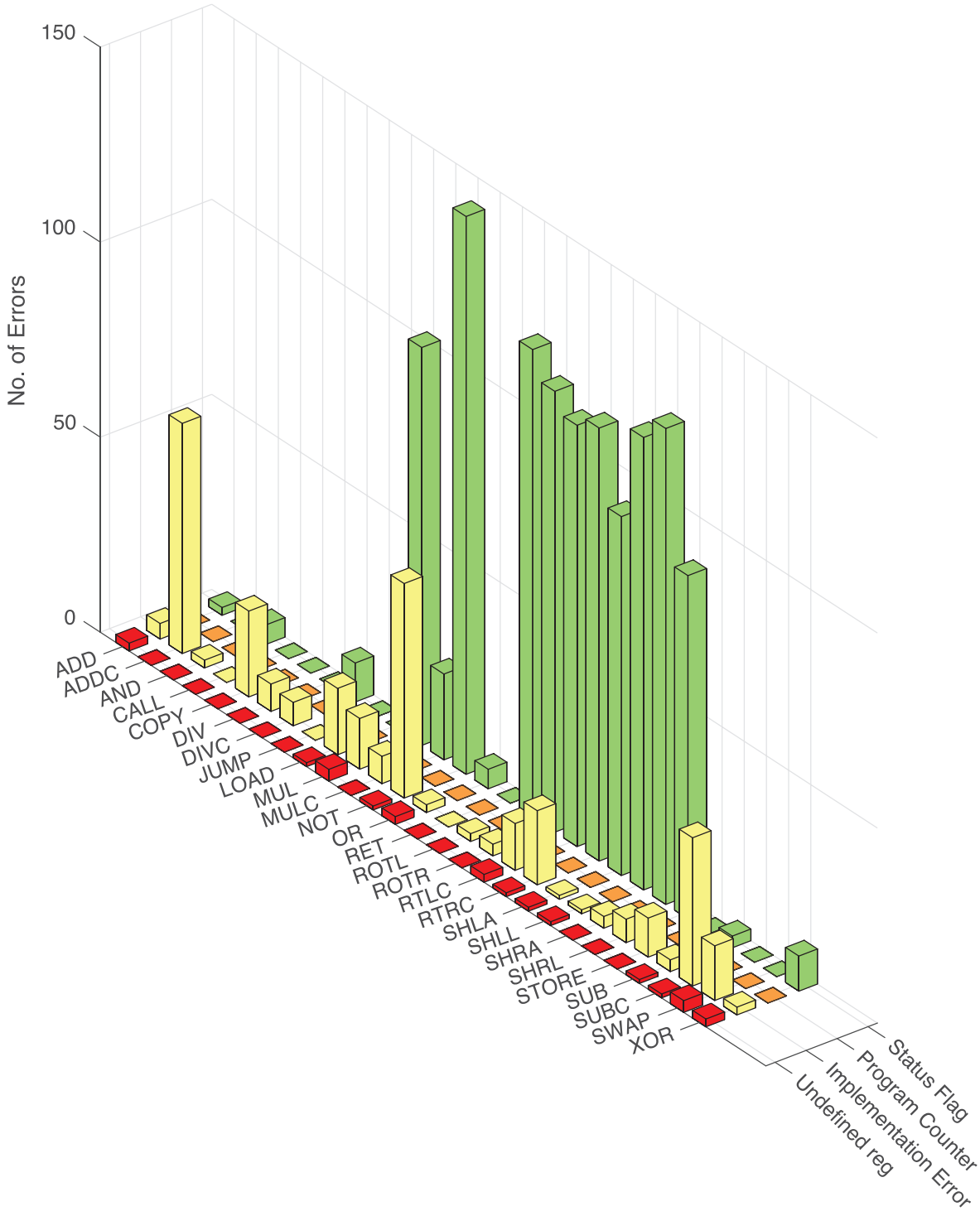

Figure V.78: Errors found in processor *sxs* while executing test *T1* (mode MD)

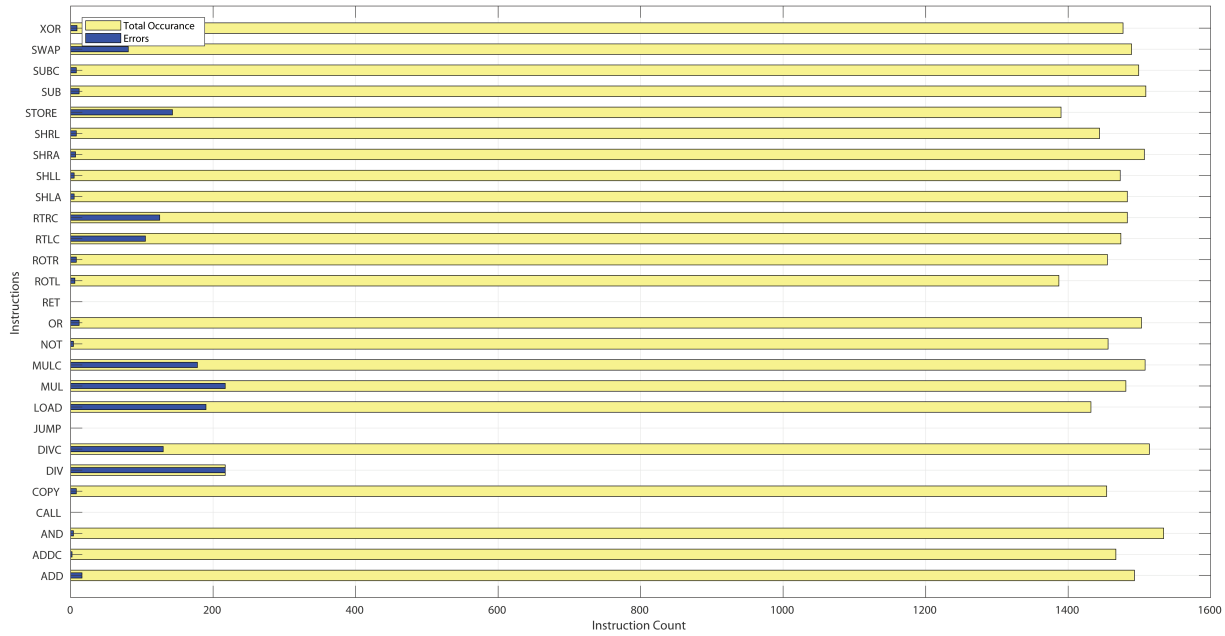

Figure V.79: Total Error count for test *T2* (mode MD) in processor *sxs*

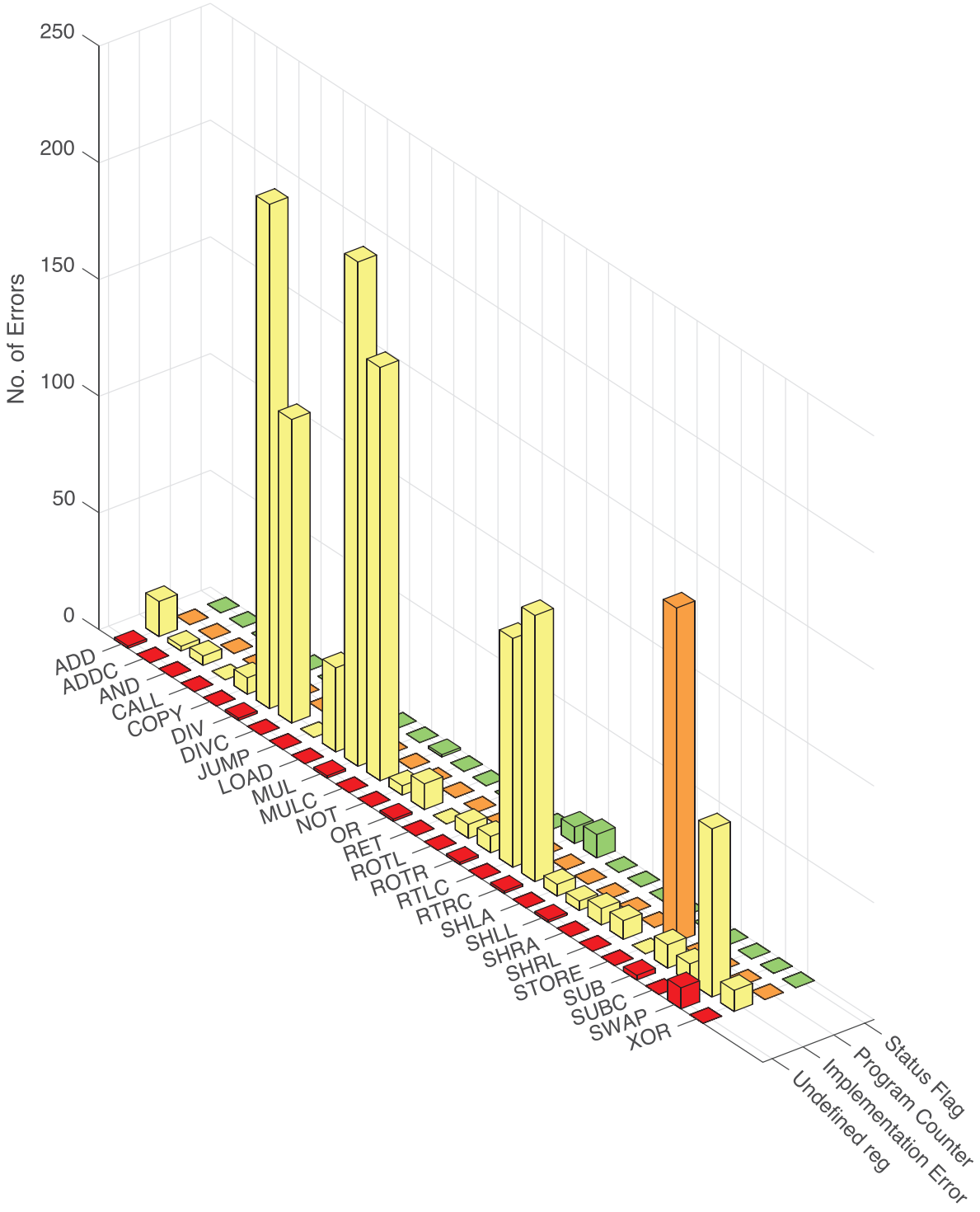

Figure V.80: Errors found in processor *sxs* while executing test *T2* (mode MD)

V.6 Processor *vxk*

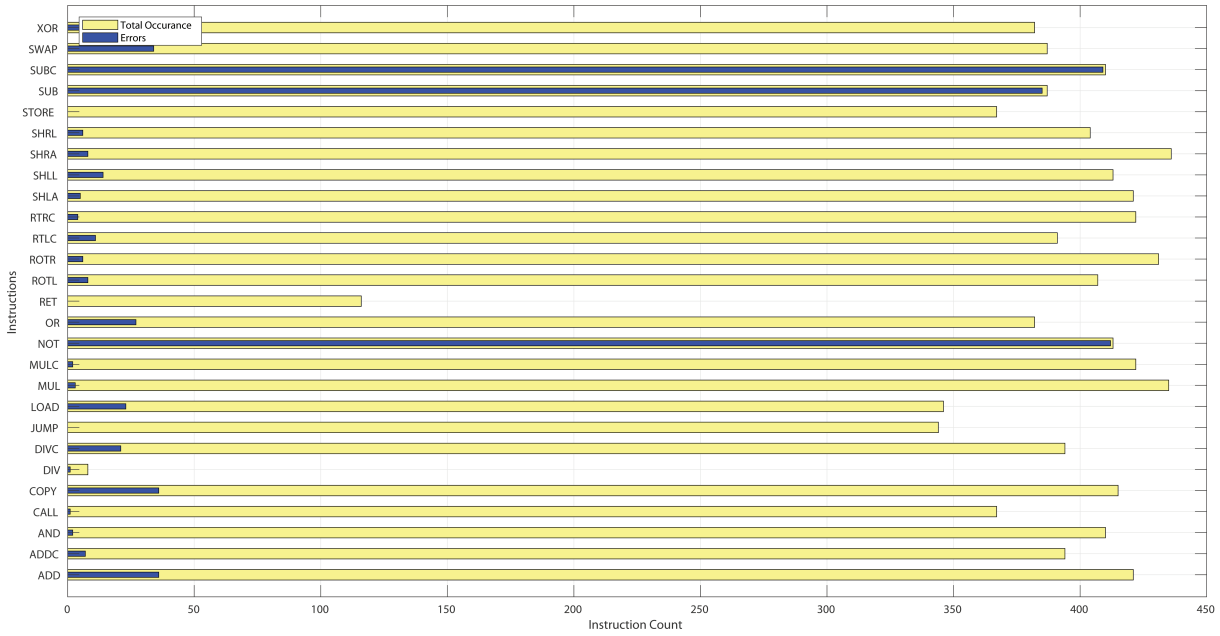

Figure V.81: Total Error count for test *T1* (mode A) in processor *vxk*

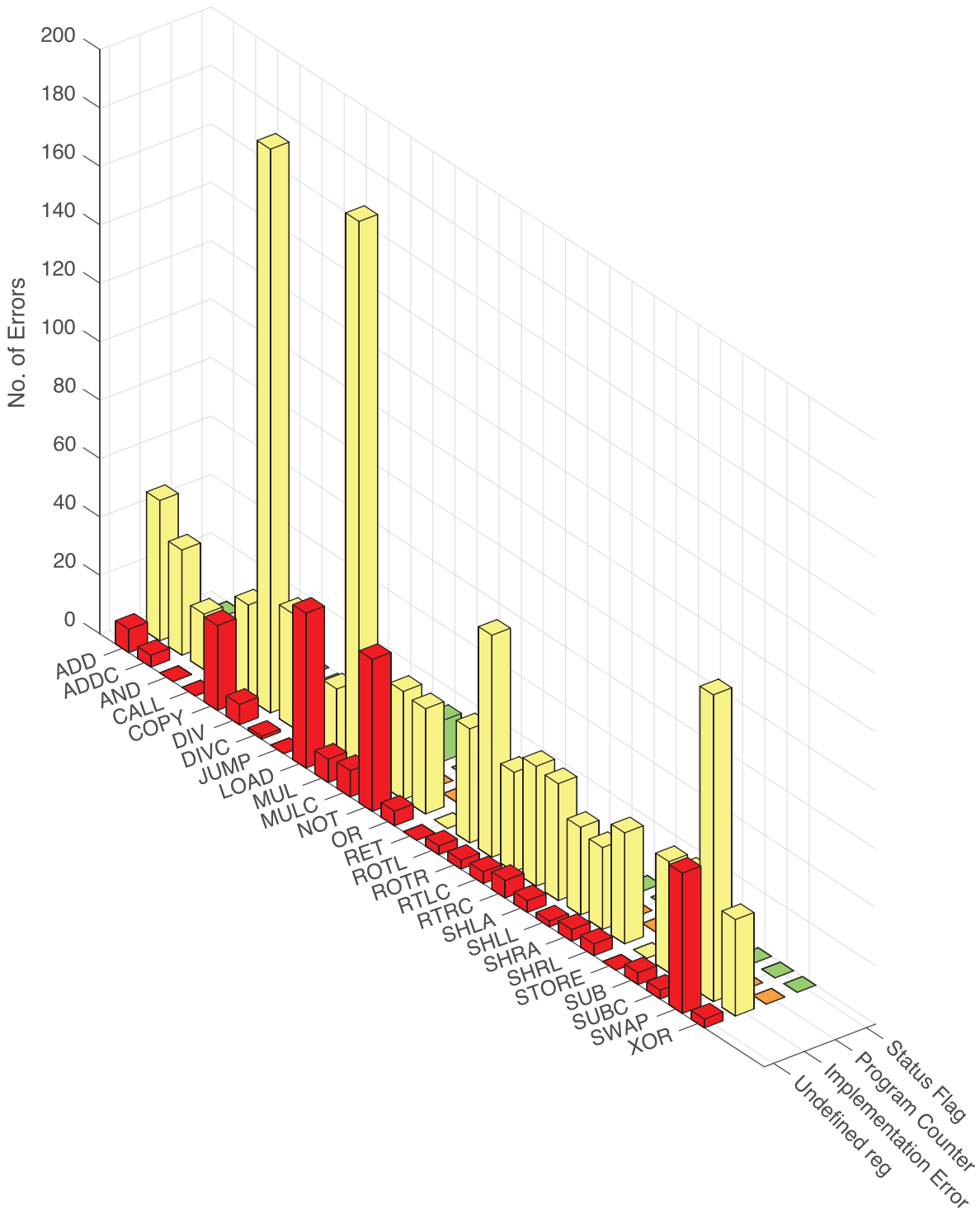

Figure V.82: Errors found in processor *vxk* while executing test *T1* (mode A)

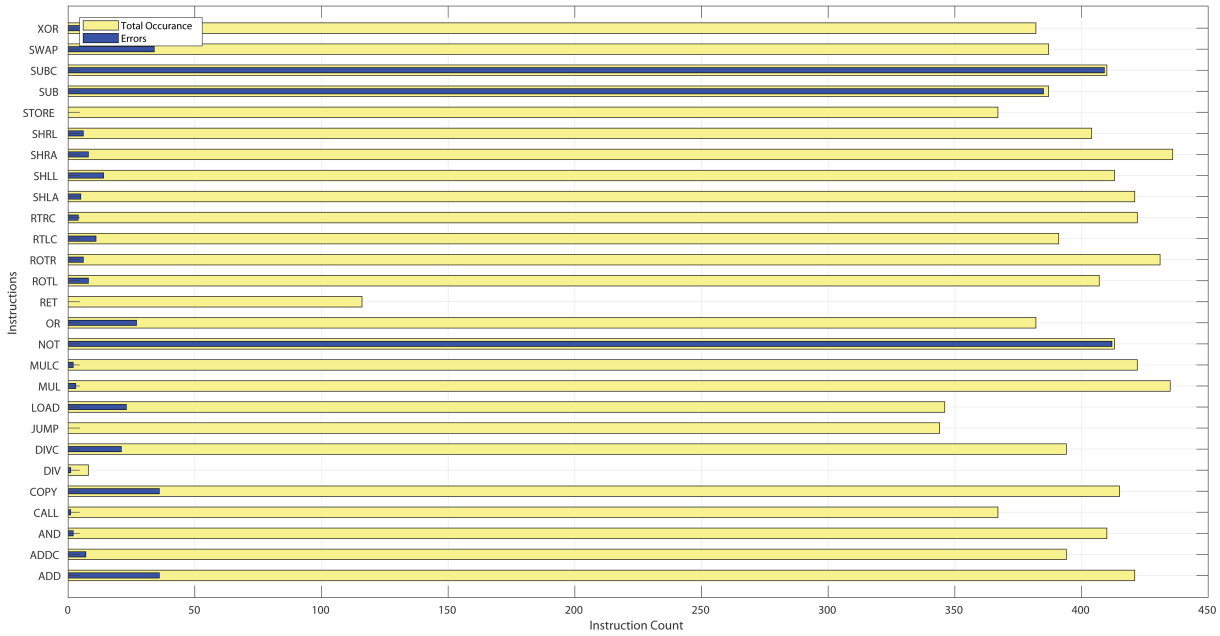

Figure V.83: Total Error count for test *T2* (mode A) in processor *vxk*

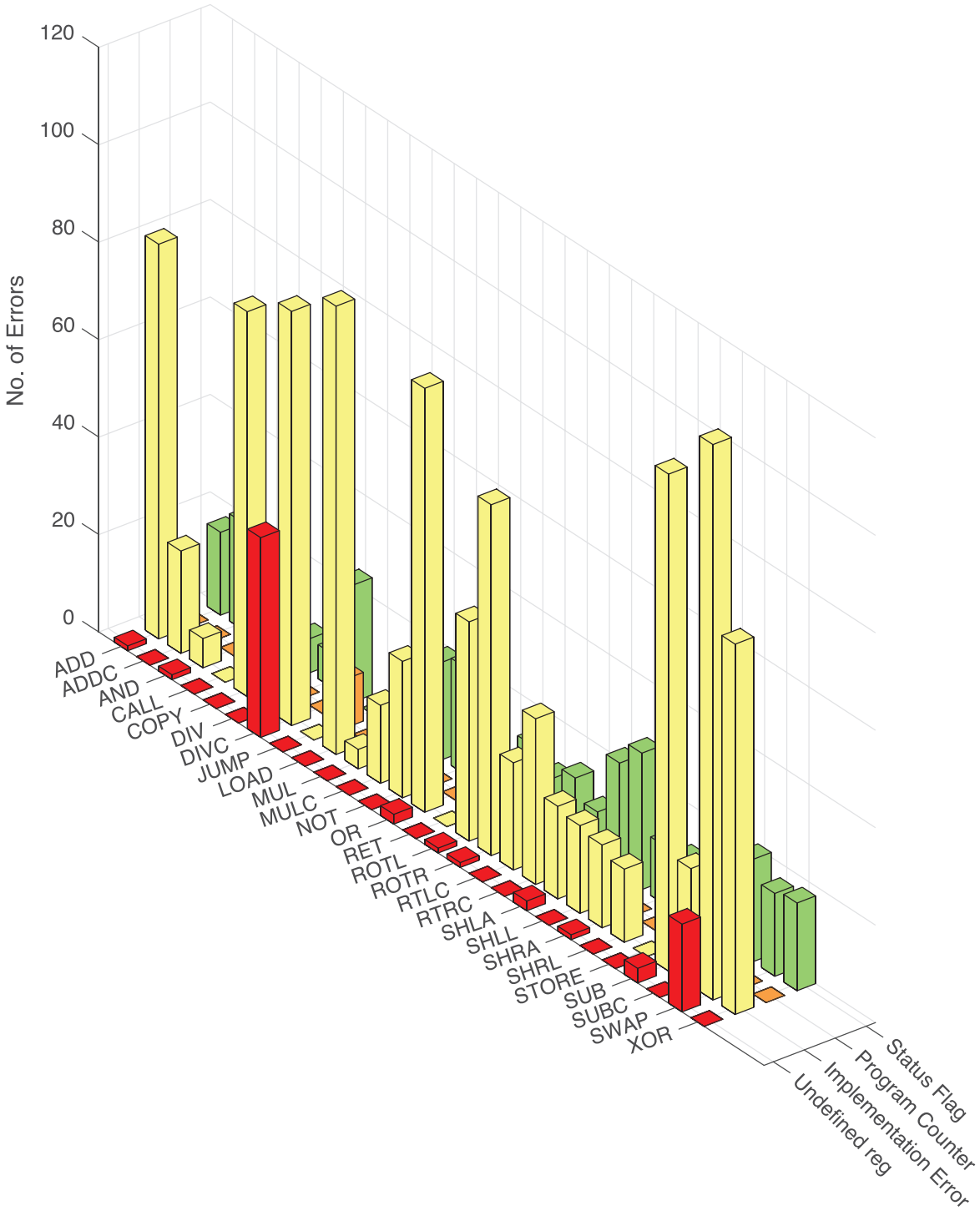

Figure V.84: Errors found in processor *vxk* while executing test *T2* (mode A)

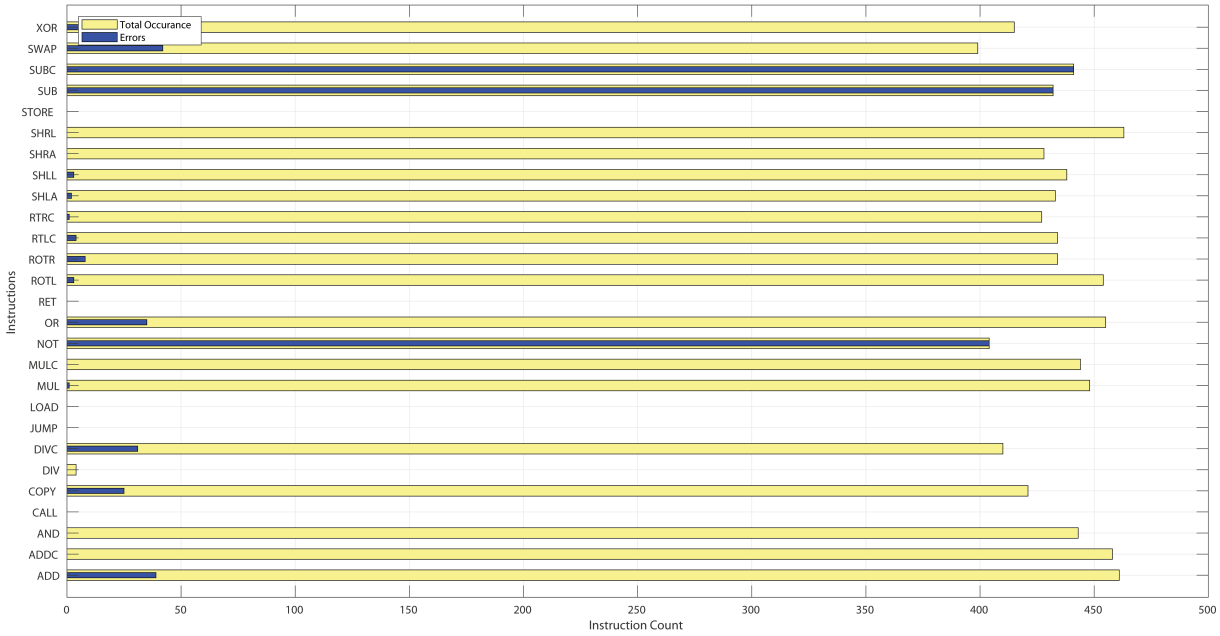

Figure V.85: Total Error count for test *T1* (mode M) in processor *vxk*

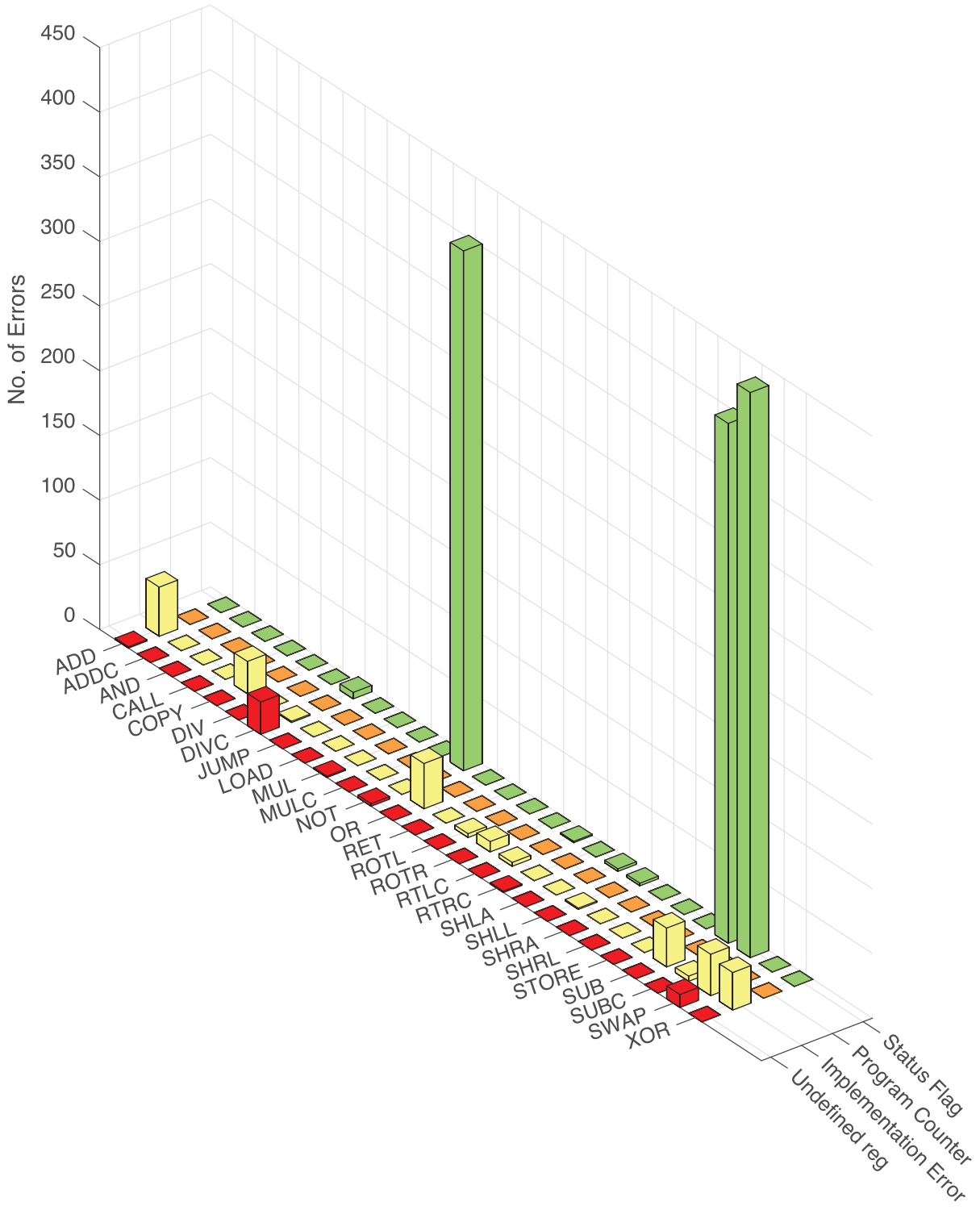

Figure V.86: Errors found in processor *vxk* while executing test *T1* (mode M)

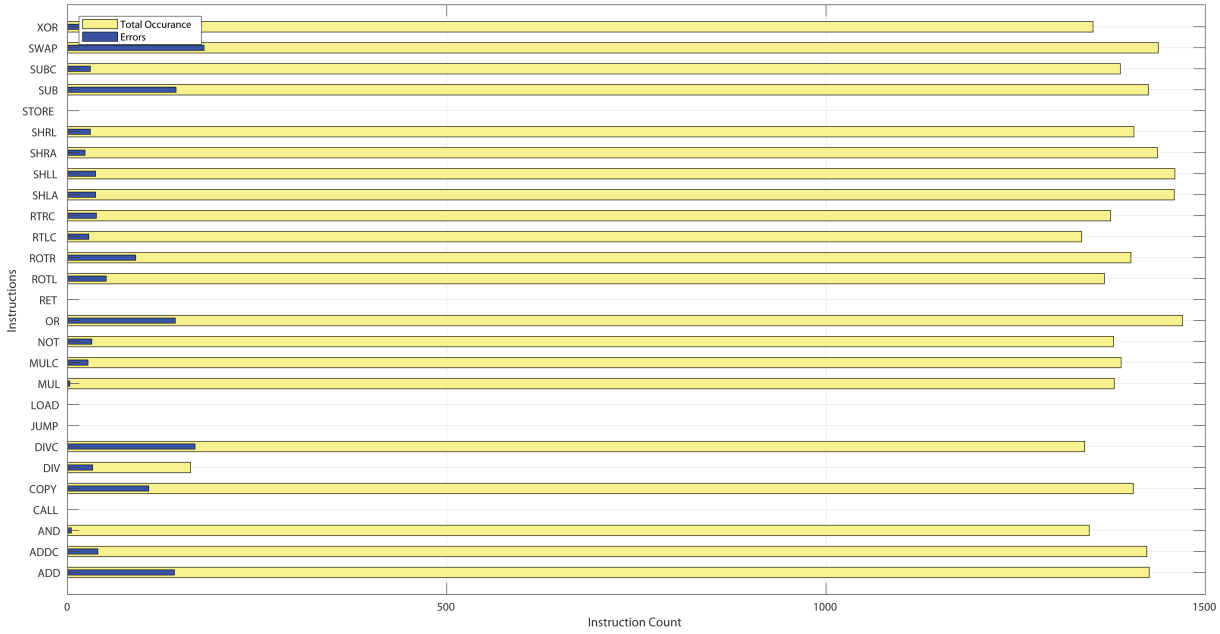

Figure V.87: Total Error count for test *T2* (mode M) in processor *vxk*

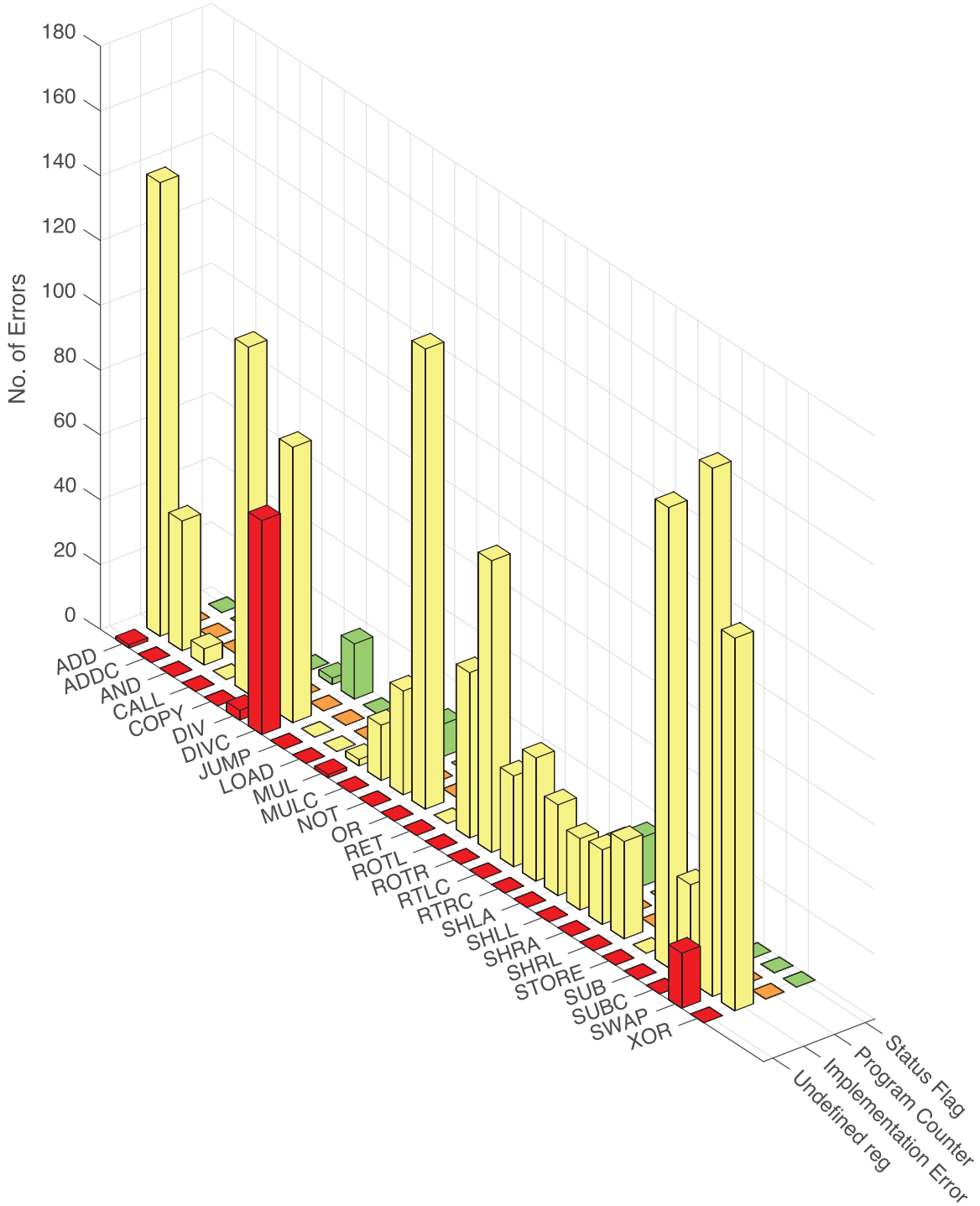

Figure V.88: Errors found in processor *vxk* while executing test *T2* (mode M)

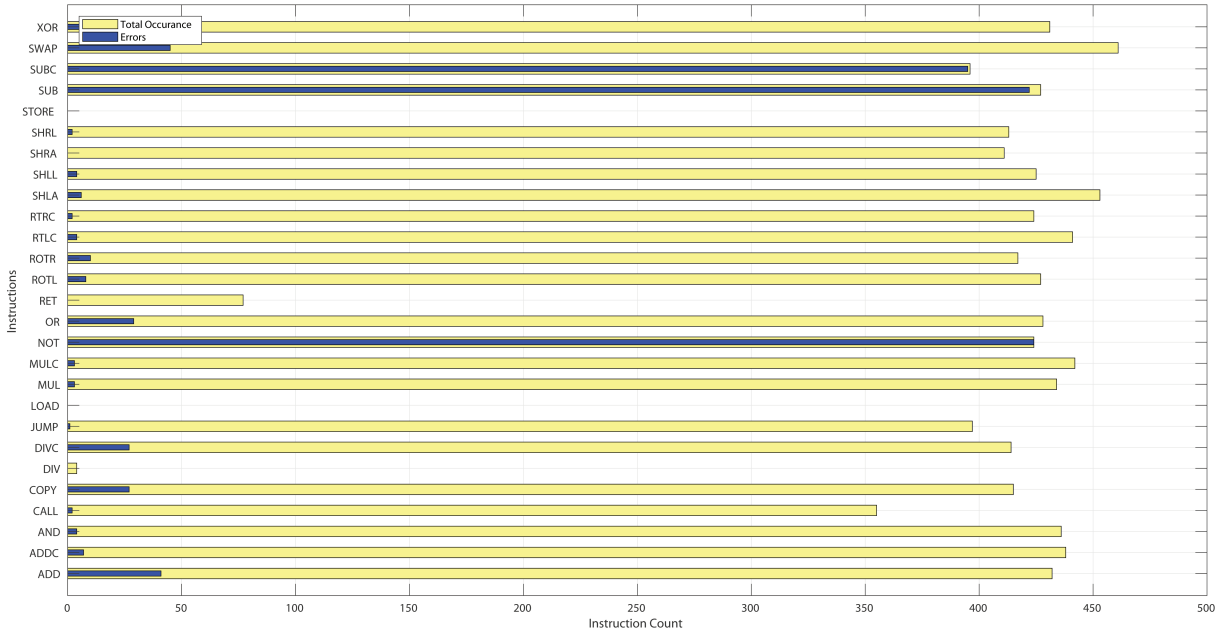

Figure V.89: Total Error count for test *T1* (mode MB) in processor *vxk*

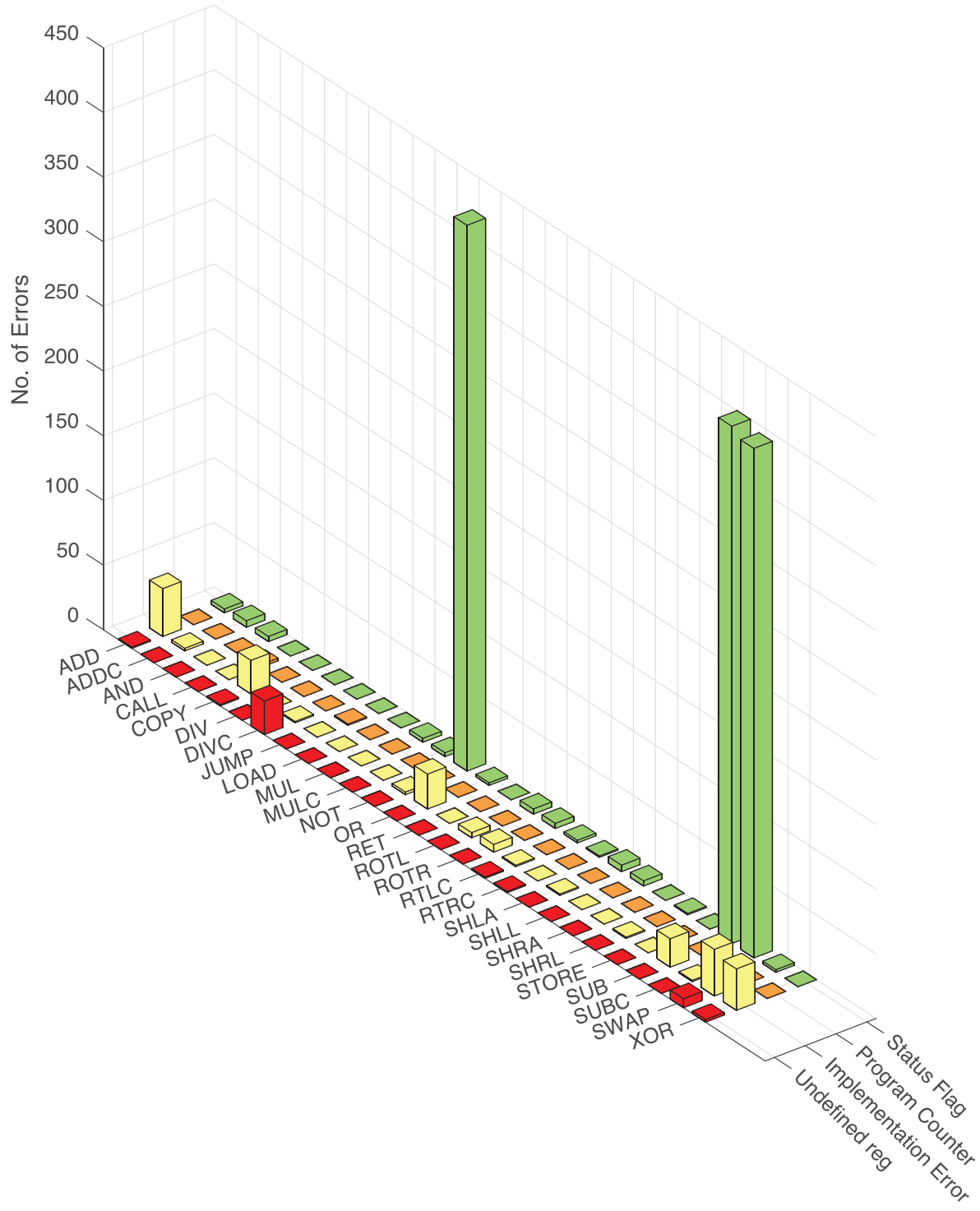

Figure V.90: Errors found in processor *vxk* while executing test *T1* (mode MB)

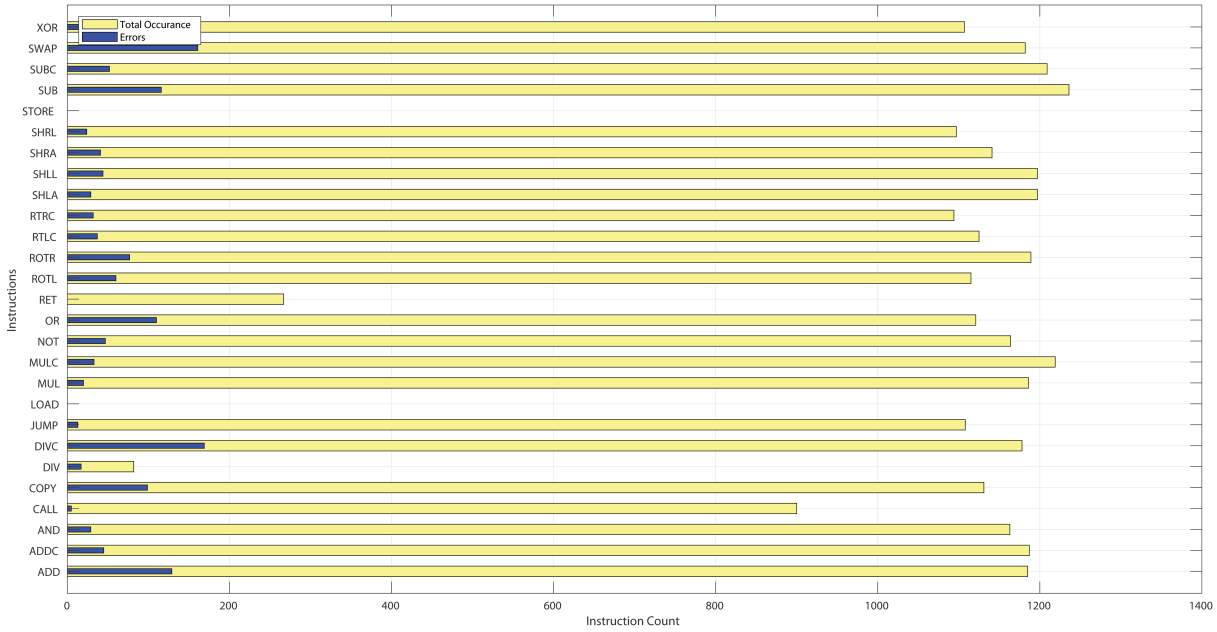

Figure V.91: Total Error count for test *T2* (mode MB) in processor *vxk*

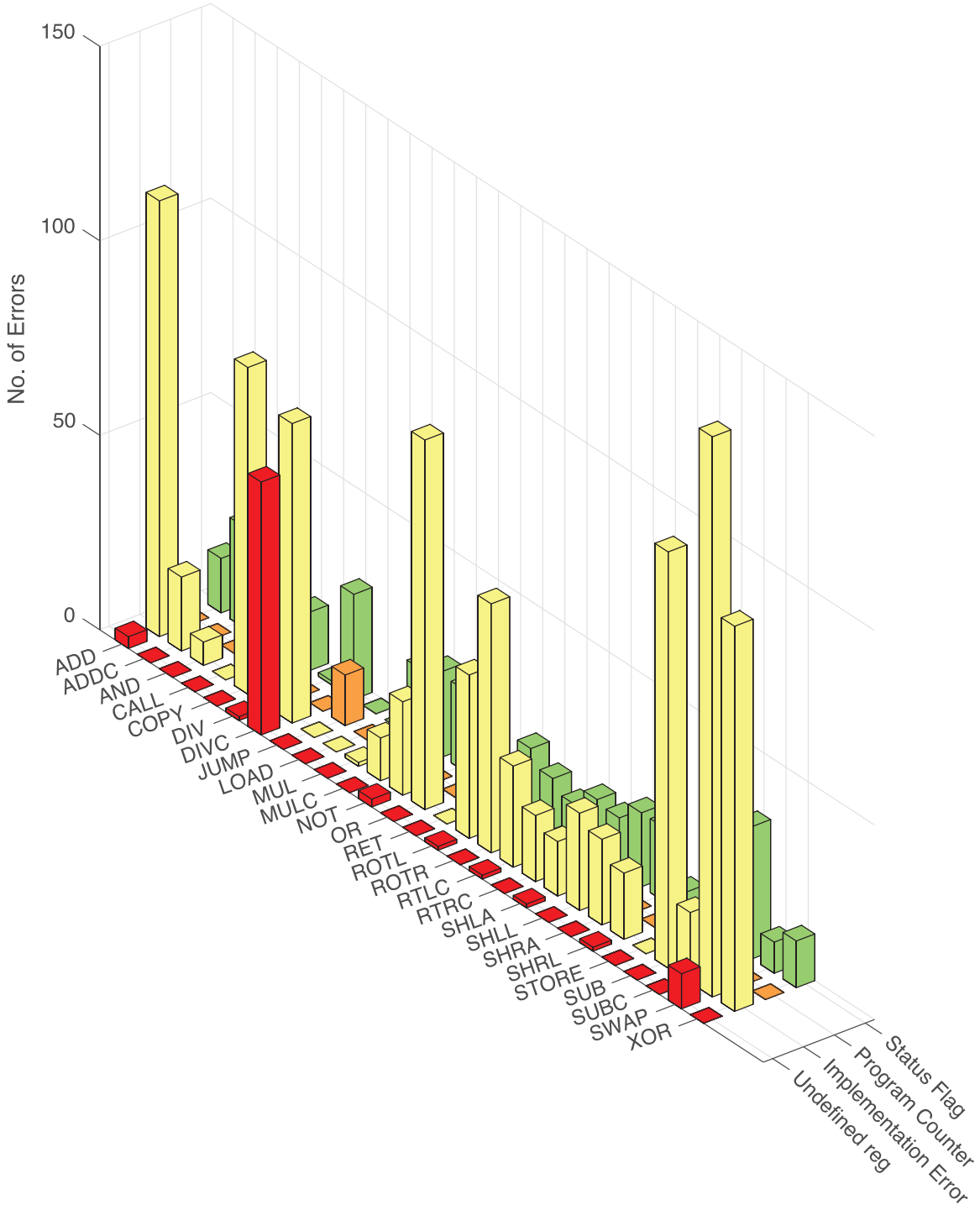

Figure V.92: Errors found in processor *vxk* while executing test *T2* (mode MB)

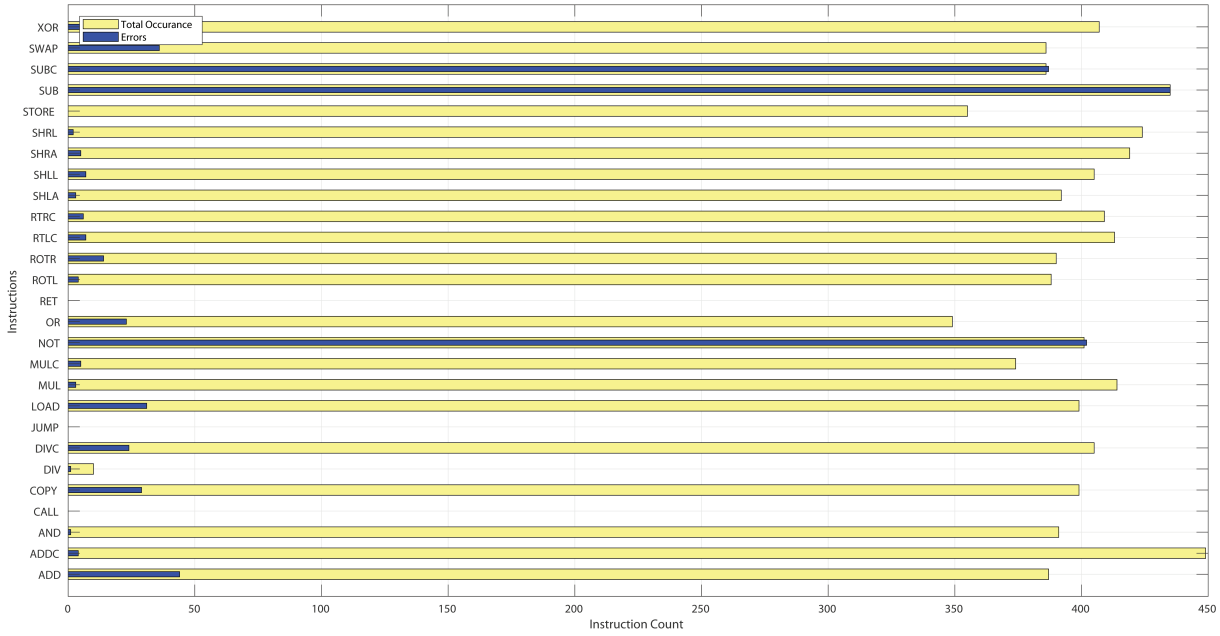

Figure V.93: Total Error count for test *T1* (mode MD) in processor *vxk*

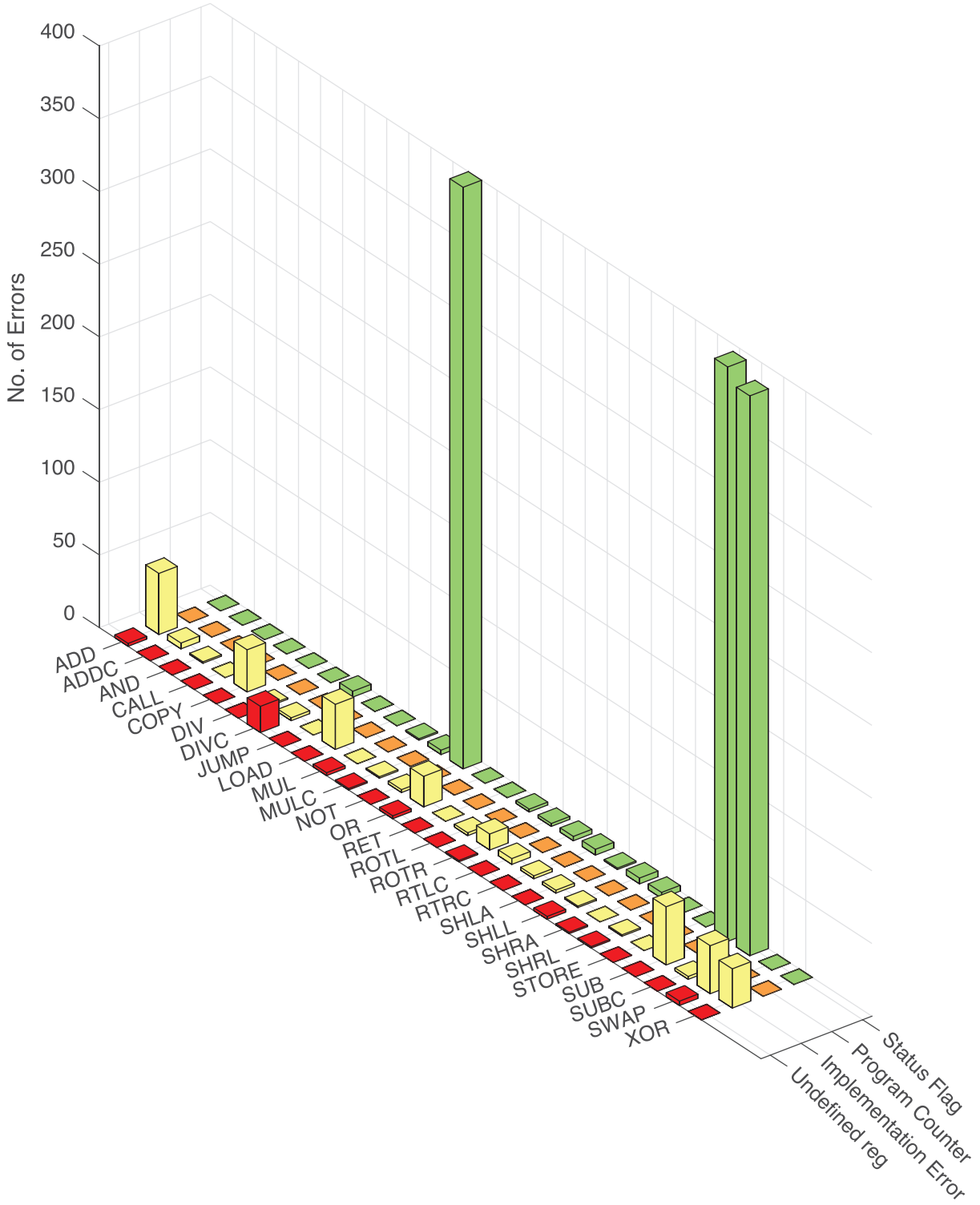

Figure V.94: Errors found in processor *vxk* while executing test *T1* (mode MD)

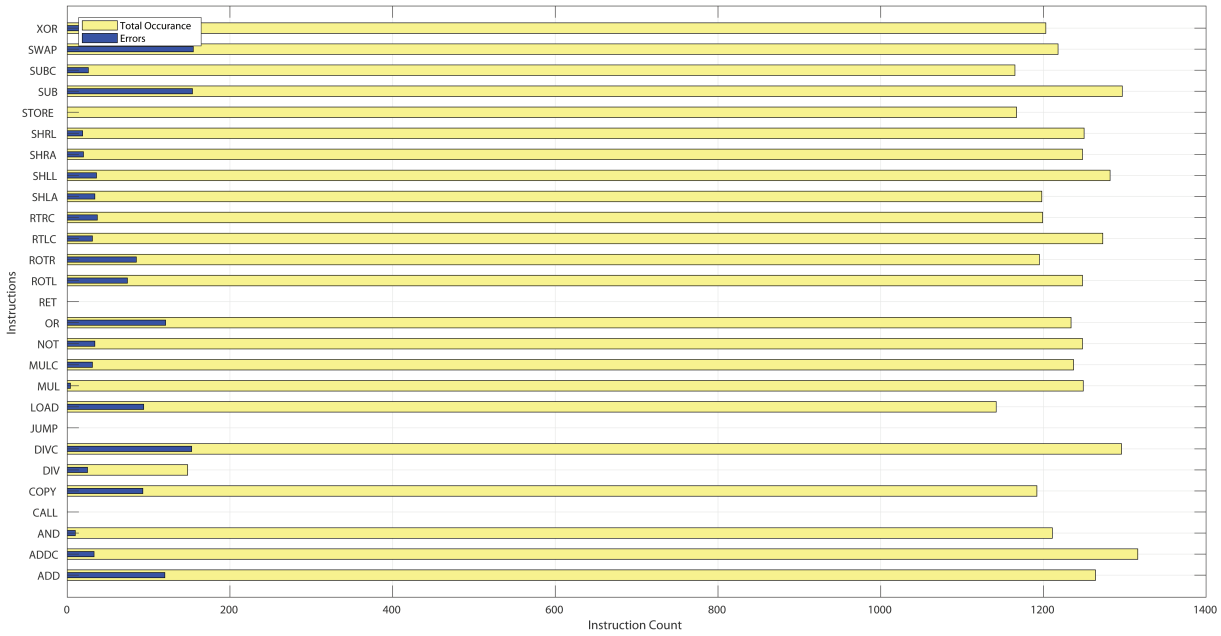

Figure V.95: Total Error count for test *T2* (mode MD) in processor *vxk*

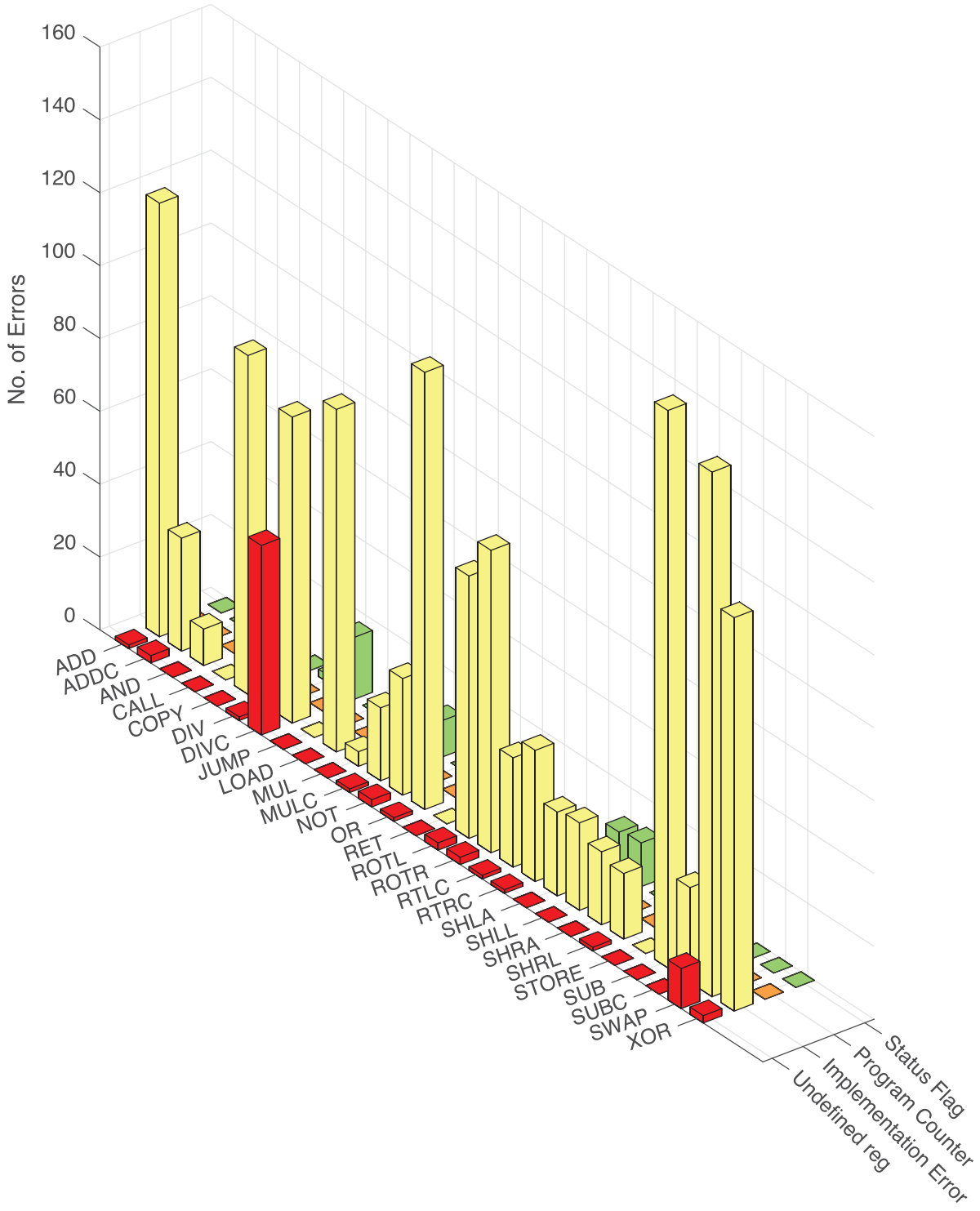

Figure V.96: Errors found in processor *vxk* while executing test *T2* (mode MD)